João Miguel Lopes de Almeida Rocha

Aceleração GPU da Animação de Superfícies Deformáveis.

Lisboa

2008

Nº do aluno: 26213
Nome: João Miguel Lopes de Almeida Rocha

Título da dissertação:
Aceleração GPU da animação de Superfícies Deformáveis

Palavras-Chave:

- GPU – processador gráfico
- GPGPU – *General Purpose GPU programming*
- Simulação de tecidos
- CUDA – *Compute Unified Device Arquitecture*
- Métodos de integração numérica
- Equações diferenciais
- MPCG – *Modified Preconditioned Conjugate Gradient*

Keywords:

- GPU – Graphic Processing Unit
- GPGPU – General Purpose GPU programming
- Cloth Simulation
- CUDA – Compute Unified Device Arquitecture
- Numerical solvers
- Diferential equations
- MPCG – *Modified Preconditioned Conjugate Gradient*

Agradecimentos

Gostava de agradecer à minha mãe pela paciência que teve comigo ao longo do ultimo ano, para não falar em todos os anteriores anos que ela já me aturou, mesmo quando, apesar de viver com ela, passava uma semana inteira sem me ver porque eu vinha para a faculdade trabalhar até de madrugada, ou ia tocar a qualquer lado com a banda. Por todo o apoio que me deu ao longo de toda a minha vida, mesmo nas opções que não compreendeu tão bem. Pela educação que me deu. Por tudo! Por estar sempre lá! Sem ela, não teria chegado aqui e não seria como sou.

Ao meu pai por sempre me ter dado na cabeça para não esquecer o curso de engª informática, mesmo apesar da minha maior paixão pela música e do meu trabalho na música. Por sempre ter sido um bom amigo. Por sempre me ter dito as coisas como elas são, mesmo que eu não gostasse. Por sempre me ter incutido o que é estabelecer prioridades. Por tudo! Sem ele, não teria, também, chegado aqui.

Aos meus colegas e amigos que estiveram comigo sempre nas horas que o neurónio vacilou um pouquinho e me puxaram para cima e motivaram. Vocês são maravilhosos! Em especial ao Nuno Prata e ao Artur Martins, colegas de “trabalho”. Obrigado!

Gostava ainda de agradecer ao professor João Lourenço. Nunca me vou esquecer da conversa que tivemos, ainda antes de eu escolher a tese, quando ponderava candidatar-me, entre outras, a uma tese sua. Sempre gostei da sua honestidade e frontalidade. Obrigado ainda pelo apoio que me deu sempre que lho pedi e ainda pela motivação extra que me deu, nomeadamente quando me desafiou a fazer-lhe uma apresentação sobre o CUDA. Obrigado!

Um grande obrigado também ao professor Pedro Medeiros, que sempre me motivou, sempre me ouviu e que foi um apoio particularmente importante em algumas fases do meu trabalho.

Mas o maior obrigado de todos vai para o meu orientador, Fernando Birra. Por tudo! Ao longo deste ano creio que não tive apenas um orientador. Tive um colega de trabalho, um companheiro! Sempre que precisei de orientação, motivação ou apoio, nunca me falhou. Quando andei meio perdido, puxou por mim. Acho que, acima de tudo, fiz um amigo e, como tal, espero também não falhar nas suas espectativas quanto ao meu trabalho. Um grande obrigado Fernando!

Se me esqueci de alguém, aqui fica um grande obrigado também! Tudo e todos os que me ajudaram, motivaram e apoiaram ao longo deste ano foram certamente importantes para que eu conseguisse cumprir com este objectivo.

Resumo

A simulação de tecidos virtuais desempenha um papel importante em diversas áreas, como as indústrias dos jogos de computador e do cinema, sendo um tópico de investigação com grande actividade.

A simulação é, normalmente, efectuada recorrendo a sistemas de partículas. Sobre as partículas são, de uma forma geral, definidas uma série de interacções com base num modelo físico de superfície, que caracteriza as propriedades do tecido, sobretudo no que diz respeito às suas deformações internas. A simulação é uma tarefa de computação extremamente intensiva graças a factores como a avaliação do modelo da superfície ou a utilização de métodos de integração numérica para a resolução do sistema de equações diferenciais que determinam a dinâmica do tecido. Qualquer destes factores depende, de forma directa, do número de partículas usado para discretizar a superfície.

Na área da computação gráfica, alguns trabalhos foram já realizados no sentido de acelerar a animação da simulação de tecidos através da programação de GPU, como em [Zel05], [Zel07] e [Den06]. O GPU moderno contém vários processadores especializados em processar grandes quantidades de dados em paralelo, apresentando uma capacidade computacional, no que toca ao número de operações de vírgula flutuante por unidade de tempo, muito superior à do CPU, sendo particularmente apropriado a problemas que possam ser expressos como computações paralelas com alta intensidade de cálculo matemático.

Neste trabalho, pretende-se contribuir com a aceleração de um simulador de tecidos com realismo acrescido, desenvolvido em [Birr07], recorrendo a um modelo de hardware e programação para GPU inovador, que o apresenta como um verdadeiro co-processador genérico ao CPU, o NVIDIA CUDA [Cud07].

As contribuições previstas estendem-se à realização de um estudo sobre as vantagens e desvantagens da utilização deste modelo quando comparado com outros, como [Zel05], [Zel07] ou [Den06], através de uma análise cuidada dos resultados obtidos, bem como quais as melhores soluções conseguidas na prática.

Abstract

Cloth simulation has an important role in several areas, like the computer games industry or the movies industry.
The simulation is usually performed through the use of particle systems. Interactions between the particles are generally defined by the use of a physical model of the surface, which is responsible for the definition of cloth's mechanical properties, specially the way it reacts to its internal deformations. The surface model evaluation, or the use of numerical integration methods to solve the differential equations that describe cloth's dynamics, are factors that turn the simulation into a highly intensive CPU task. Both these factors are directly dependent on the number of particles used to describe the surface.
In the computer graphics community, some efforts have been made in order to speed up cloth simulation through general programming on the GPU, like in [Zel05], [Zel07] or [Den06]. Modern GPUs contain lots of processing units specialized in the parallel processing of large amounts of data, presenting a lot more computational «horsepower» than the modern CPU, relating to single precision floating point operations per time unit. Modern GPU, then represents a specialized unit for the parallel processing of massive mathematic calculations.
In this work, the main contribution will be the speed up of an increased realism cloth simulator, proposed in [Birr07], using a new programming model that allows us to explore the whole computational power of the GPU and its use as a general co-processor to the CPU, the NVIDA CUDA [Cud07].
A wider set of contributions will be the study of the advantages or disadvantages on the use of this programming model when compared with previous results in the area, like [Zel05], [Zel07] or [Den06], through the careful analysis of the attained results, as well as what were considered the best practical solutions.

Índice

Índice de imagens

1. Introdução

A investigação na animação de superfícies deformáveis é uma área com múltiplas aplicações, como nas áreas de saúde, indústria e de animação gráfica, entre outras. Este trabalho insere-se num caso particular das superfícies deformáveis, os tecidos.

A simulação de tecidos é uma área específica do estudo de superfícies deformáveis com grande actividade e cuja maturidade é inegável. Os modelos que definem a estrutura da superfície, a forma como se codificam as interacções internas do tecido, e as técnicas usadas para fazer avançar o sistema dinamicamente, são áreas que estão em permanente desenvolvimento e evolução.

Sobre os modelos usados para a estrutura interna do tecido, reconhecem-se actualmente as vantagens da utilização de sistemas de partículas como ferramenta de modelação da superfície. A investigação associada à dinâmica dos tecidos nasce com a aplicação de métodos de integração numérica para integrar as trajectórias das partículas. Métodos de integração explícitos e implícitos surgem em diversos trabalhos nesta área, sendo reconhecidas as técnicas implícitas como as que permitem, simultaneamente, maior estabilidade de simulação e passos de integração maiores.

Apesar desta maturidade evidente, o grau de realismo está ainda muito longe de ser considerado satisfatório. A complexidade dos fenómenos associados aos tecidos é tal que requer um poder computacional muito elevado. Além disto, não existem técnicas capazes de aliar processos mecânicos macroscópicos, como a elasticidade e a curvatura, aos processos mecânicos microscópicos, como o desgaste ou as alterações produzidas por mudanças atmoféricas ou climáticas, sendo que os modelos actuais se resumem, então, apenas a um destes dois níveis de detalhe. No contexto da computação gráfica, onde este trabalho se insere, proliferam os modelos dedicados às propriedades macroscópicas.

1.1 Motivação

A motivação principal que serviu de impulso para o início dos trabalhos desta dissertação foi a ideia de acelerar um simulador de tecidos com realismo acrescido [Birr07] tirando partido da capacidade computacional do GPU moderno. Os modelos de tecidos mais modernos são extremamente complexos e o uso criterioso do poder de computação é uma mais valia, especialmente quando a simulação é um processo moroso, de complexidade temporal não linear no que toca ao número de partículas usadas, como é o caso.

A introdução recente de processadores multi-core, colocando processadores em paralelo num único *chip*, permitiu aos fabricantes de CPUs melhorias de performance mantendo as mesmas frequências de processamento. Durante a ultima década, uma aproximação semelhante foi tomada pelos fabricantes de processadores gráficos (GPUs), sob influência da indústria dos jogos e na tentativa de fornecer simulações cada vez mais detalhadas e realistas. Esta abordagem levou a uma tal evolução da capacidade de processamento das placas gráficas que o GPU moderno apresenta performances muito superiores às do CPU, no que toca ao número de operações de vírgula flutuante efectuadas por unidade de tempo. (Figura 1). Só nos últimos sete anos, o GPU evoluíu de uma peça de hardware de funcionalidades fixas sobre primitivas gráficas, para uma peça com um processador paralelo programável, com performances muito superiores às do CPU moderno, em particular no que toca a operações vectoriais paralelas. Mais especificamente, o GPU moderno é particularmente apropriado a problemas que possam ser expressos como computações paralelas com alta intensidade de cálculo matemático, pois contém vários processadores especializados em processar grandes quantidades de dados em paralelo. Para ter uma ideia do porquê desta especialização, basta pensarmos, por exemplo, que sendo o principal objectivo de uma placa gráfica «pintar» pixeis no ecrãn, com as resoluções actuais, é necessário pintar milhões de pixeis por cada *frame*.

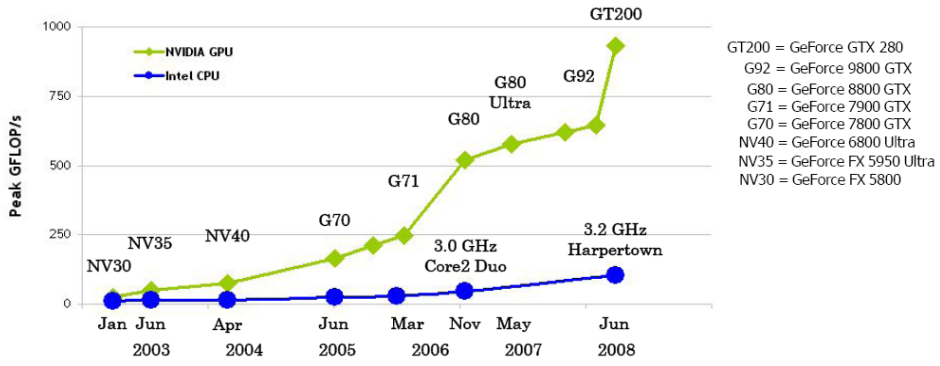

Figura 1 – evolução da capacidade de processamento do GPU face à mesma evolução no CPU, entre Janeiro de 2003 e Junho de 2008, NVIDIA.

A ideia de tirar partido deste poder de processamento para acelerar a animação da simulação de tecidos é atraente e foram já apresentados diversos trabalhos neste sentido, como [Zel05], [Zel07] e [Den06]. No entanto, tirar partido de toda esta capacidade de processamento está longe de ser trivial. Inicialmente, a programação do GPU tinha de ser feita em linguagem de baixo nível e era necessário um conhecimento muito específico do *hardware*. Novas gerações de GPUs ofereciam mais possibilidades e surgiu a necessidade de introduzir linguagens de mais alto nível. Com a introdução de linguagens de *shading* como o GLSL [Kes07], foi dada a possibilidade aos programadores de definir o que acontece ao nível de fases específicas da síntese gráfica, mais especificamente, ao nível do processamento dos vértices (*vertex shading*) e ao nível do processamento de pixeis (*pixel shading*). Esta é a abordagem tomada em [Zel05], [Zel07] e [Den06], no âmbito da aceleração da animação da simulação de tecidos. Os problemas são mapeados para problemas gráficos onde os dados são representados como pixeis coloridos em texturas. *Shaders* são depois aplicados para efectuar computação sobre os dados.

Este tipo de abordagem implica lidar com APIs gráficas o que apresenta problemas como, por exemplo, limites no tamanho de texturas – referido em [Den06] como uma das limitações do trabalho desenvolvido. Outro tipo de problema relevante nesta abordagem tem que ver com o facto dos *shaders* poderem ler de forma genérica da memória gráfica mas não poderem escrever de forma genérica, o que retira, no GPU, muita da flexibilidade de programação disponível em CPU.

O CUDA [Cud07] é um novo modelo de programação e de *hardware*, lançado pela NVIDIA, que responde directamente a estas limitações expondo o GPU como um verdadeiro co-processador paralelo genérico. O modelo de programação pode ser encarado como uma extensão ao C. O GPU é visto como uma unidade de processamento que serve de co-processador ao CPU ou *host*, que tem a sua própria memória, e que corre muitos *threads* em paralelo. Idealmente, porções paralelizáveis de uma aplicação são, então, executadas no co-processador como programas – *kernels* – que correm em paralelo em muitos *threads*.

A ambição deste trabalho prende-se, assim, com o objectivo de tentar tirar o máximo partido das capacidades de computação do GPU, recorrendo ao modelo de programação CUDA, para acelerar o simulador de tecidos com realismo acrescido proposto em [Birr07].

1.2 Âmbito do trabalho

O âmbito deste trabalho é então, avaliar a possibilidade de acelerar o trabalho desenvolvido em [Birr07] recorrendo ao modelo de hardware e de programação fornecido pelo CUDA.

Em [Birr07], procedeu-se à implementação do simulador de tecidos com realismo acrescido, baseado no modelo de tecidos proposto por Baraff e Witkin em [Bar98], sendo que, provavelmente, a maior contribuição seja a aplicação de uma técnica de variação do nível do detalhe durante a simulação, que implica que o número total de partículas no sistema varie dinamicamente com o decorrer da simulação.

A aplicação desenvolvida em [Birr07] pode dividir-se em três fases principais:

- A avaliação do modelo de tecidos (*Evaluation*), onde são determinadas as resultantes das forças, internas e externas, sobre cada partícula do sistema.
- A execução do passo de integração (*Solve*), que inclui a construção das matrizes das derivadas parciais e onde é resolvido o sistema de equações lineares gerado pela avaliação do modelo implícito.
- A variação do nível do detalhe (*DLOD*), que inclui a aplicação da técnica de variação do nível de detalhe bem como as tarefas de gestão das estruturas de dados associadas aos objectos que vão sendo criados, destruídos, activados ou desactivados, no decorrer da simulação. Note-se que, não havendo variação do nível de detalhe, este conjunto de tarefas é desnecessário, pois não há mudança alguma no conjunto de objectos usados durante toda a execução.

Se olharmos para a figura (Figura 2), é fácil verificar que a simulação de tecidos utilizando o método de integração implícita de Euler é dominada, quase na totalidade, pelo próprio método de integração numérica, no que toca a tempo de processamento. Mesmo com a aplicação da técnica de variação de nível de detalhe em [Birr07], é visível que, tanto o peso desta técnica (*DLOD*), como a própria avaliação do modelo (*Evaluation*), são consideravelmente inferiores ao peso computacional envolvido na resolução do método de integração.

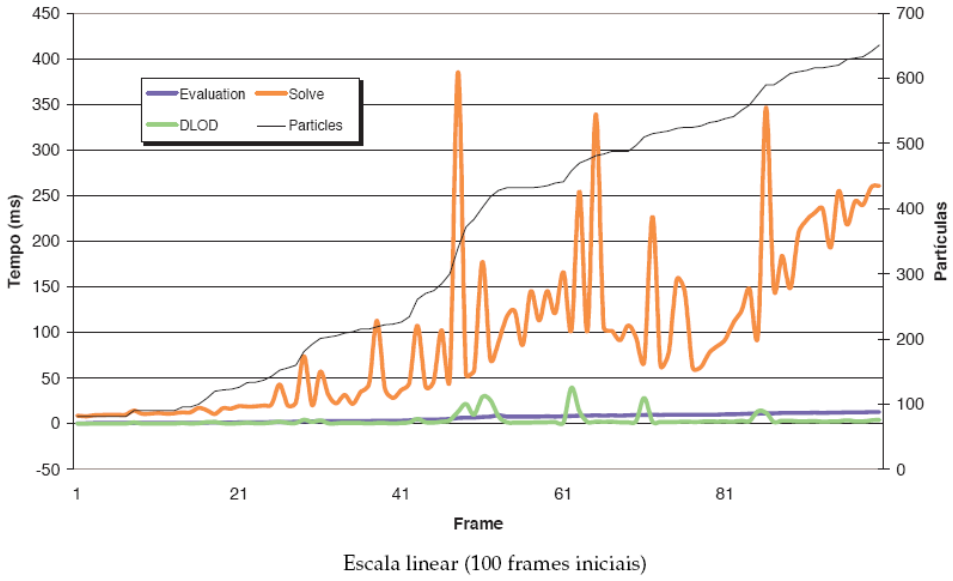

Figura 2 - Perfil do tempo de execução gasto nas diversas etapas da simulação em [Birr07].

Um outro resultado que é visível no gráfico da figura (Figura 2) é o impacto que a técnica de variação do nível de detalhe produz no método de integração. A linha *Solve* apresenta picos bruscos que correspondem a momentos em que a integração implícita com passo adaptativo é forçada a reduzir o passo de integração, por forma a manter a estabilidade. Nota-se uma relação directa entre o crescimento mais brusco do número de partículas e os referidos picos na curva da resolução do sistema.

Um dado importante, referido em [Birr07] e que não está tão evidente neste gráfico, é a penalização que a técnica de gestão dinâmica do detalhe provoca, de forma sistemática, em todos os passos de simulação e que se traduz na necessidade de recriar a estrutura das matrizes das derivadas parciais. Usando uma malha de resolução fixa, sem variação no número de partículas, dado que a topologia da mesma é constante, as entradas nas matrizes das derivadas parciais são as mesmas em todos os instantes da simulação, pelo que entre cada passo de simulação bastará actualizar os respectivos valores. No entanto, no caso da simulação com detalhe dinâmico, novas partículas podem ser introduzidas ou removidas. Partículas que desaparecem podem deixar posições vazias nas referidas matrizes, caso a taxa com que são destruídas seja superior à taxa de criação de novas partículas. Isto implica que, na implementação em [Birr07], estas matrizes são reconstruídas a cada passo, caso haja pelo menos uma operação de simplificação ou subdivisão.

1.3 Abordagem e contribuições previstas

A abordagem que se propõe é, então, a de passar a execução do passo de integração (*Solve*), que envolve a resolução do sistema de equações lineares, para execução paralela em GPU, recorrendo ao modelo de *hardware* e de programação CUDA.

As principais contribuições previstas são então:

- Utilização de um modelo de *hardware* e de programação para GPU inovador, que esconde algumas da limitações da programação em GPU por APIs gráficas, apresentando o GPU como um verdadeiro co-processador genérico.
- Desenvolvimento de um algoritmo, recorrendo a este modelo, para criação de matrizes esparsas e resolução de sistemas lineares de equações, especialmente desenhado para que possa ser executado em paralelo no processador gráfico com um mínimo de intervenção do processador central (CPU).
- Realização de um estudo sobre as vantagens e desvantagens da utilização desta abordagem perante outras, através de uma análise cuidada dos resultados obtidos, bem como quais as melhores soluções conseguidas na prática.

2. Estado da Arte

O gesto simples de pendurar uma toalha num gancho de parede não faz antever a complexidade inerente aos processos físicos que ocorrem para que essa toalha fique em repouso. Um observador mais atento poderá talvez reparar em pormenores como o esticar da toalha na zona circundante ao ponto de apoio ou o enrugar da toalha, com vincos mais acentuados que partem geralmente desse ponto. Tendo em conta que estamos apenas a falar de efeitos macroscópicos e visíveis, podemos então ter uma ideia de quão complexo é descrever todo este processo a um nível mais detalhado, como por exemplo para simulação.

A simulação do comportamento dinâmico de tecidos é um tópico de investigação complexo e com grande actividade. Os tecidos são um tipo específico de superfícies deformáveis, cuja modelação é bastante mais complexa do que a de superfícies rígidas, uma vez que os graus de liberdade de uma superfície deformável são incomparavelmente maiores que os de uma superfície rígida.

Existem fundamentalmente duas comunidades distintas com interesse na investigação sobre modelos de simulação de tecidos: a comunidade ligada à industria têxtil e a comunidade gráfica. Apesar dos estudos em ambas terem, no geral, objectivos diferentes, há uma necessidade comum de desenvolver modelos matemáticos que permitam a simulação de variados aspectos dos tecidos como a sua dinâmica, aparência ou até propriedades mecânicas de determinados tipos específicos de tecido. Para a comunidade gráfica, na qual se insere o conteúdo desta tese, a ênfase situa-se mais no realismo visual, partindo-se do princípio de que uma simulação é correcta se assim o parecer visualmente e descartando para segundo plano o rigor na modelação física, de acordo com as propriedades mecânicas do tecido. No caso da comunidade têxtil a correcção física é essencial e determinante, sendo que o comportamento dinâmico do tecido terá sempre que respeitar as propriedades físicas e mecânicas do respectivo modelo. Para um resumo muito extenso das contribuições de técnicas de modelação provenientes da comunidade têxtil, que se situa fora do âmbito deste trabalho, aconselha-se a consulta de [Bre00].

2.1 Modelos de tecidos

No contexto da comunidade gráfica, em que este trabalho se insere, existem fundamentalmente três formas de modelar tecidos: através de modelos geométricos, modelos híbridos ou modelos físicos.

As técnicas de modelação geométrica têm como principal característica a maior preocupação pela aparência dos tecidos, em especial a forma, as dobras e os vincos. Para atingir este objectivo, estas características são representadas através de equações geométricas e, como tal, as propriedades reais do tecido não são simuladas de todo. Esta abordagem, na realidade, apenas resolve o problema da visualização de tecidos e não o da simulação do seu comportamento dinâmico – a animação. Por outro lado, na modelação geométrica os algoritmos podem ser extremamente eficientes, não sendo necessária grande capacidade computacional, dada a necessidade de efectuar um conjunto pequeno de cálculos.

A animação através de modelação física, por sua vez, implica um peso computacional elevado e as técnicas de modelação híbrida surgiram numa altura em que este peso era demasiado alto para tornar a simulação viável, com base apenas num modelo físico. O objectivo era, então, melhorar o desempenho efectuando parte da simulação física no princípio ou no final da simulação, consoante a técnica específica. O resto da simulação era feita através de aproximações geométricas, computacionalmente mais moderadas, ganhando assim em tempo de execução.

Com o poder computacional disponível actualmente, já se torna, no entanto, possível a exploração do benefício inerente à simulação segundo modelos físicos, pelo que não existe interesse da nossa parte num aprofundar das técnicas de modelação geométrica e híbrida. Como tal, para um resumo mais alargado sobre estas contribuições, sugere-se a consulta de [Bre00].

A modelação física é, assim, a técnica de modelação que maior importância adquire. Existem, fundamentalmente, duas abordagens na modelação física de tecidos:

- modelos discretos: recorrendo essencialmente a sistemas de partículas, tal como sugerido pela primeira vez para a modelação de tecidos no modelo apresentado por Breen e House [Bre92];
- modelos contínuos: como sugerido pela primeira vez para modelação de tecidos no trabalho de Terzopoulos [Ter87].

O modelo contínuo expressa a energia de deformação da superfície recorrendo a equações diferenciais, que são contínuas ao longo de toda a superfície a modelar, ao contrário do que acontece com modelação por sistemas de partículas, onde apenas se avalia o modelo em locais discretos espalhados ao longo da superfície do tecido. A grande vantagem de representar uma superfície segundo um modelo contínuo resulta do facto de que estas equações diferenciais podem ser extrapoladas directamente das propriedades mecânicas que a superfície tem na realidade. No entanto, a modelação de superfícies segundo modelos contínuos apresenta algumas limitações no tratamento de colisões e, na modelação de tecidos, as colisões são um fenómeno frequente. Será interessante a leitura de [Ter87] para uma visão mais ampla sobre este assunto.

Com um sistema de partículas, a modelação física passa quase sempre pelo uso de malhas poligonais triangulares ou formadas por quadriláteros, que representam, normalmente, um esquema de "massas e molas". Em cada vértice encontra-se um ponto com massa que aproxima determinada área do tecido e que é representado por uma partícula no sistema. Quer se usem forças ou energias, os valores destas grandezas são discretizados ao longo da superfície do tecido, sendo apenas calculados para aqueles pontos (cada partícula). Os cálculos efectuados para as forças que actuam por via das molas baseiam-se em relações de adjacência envolvendo partículas vizinhas. O número de partículas que constitui cada relação de adjacência, bem como a teoria subjacente ao cálculo, varia de técnica para técnica, podendo estas relações ser baseadas em meras equações de geometria local, ou envolver cálculos mais complexos baseados em teorias físicas conhecidas.

A simulação física pode incidir num método de minimização de energia, tendo como objectivo obter a forma do tecido numa configuração estática e estável. Pode, também, ser baseada em sistemas de equações diferenciais que, uma vez integradas, permitem actualizar as posições das partículas do sistema ao longo de intervalos de tempo de simulação.

A primeira abordagem está fora do interesse de estudo pois, na realidade, apenas apresenta uma solução visualmente realista no fim da simulação, não existindo nenhuma garantia de que nos passos intermédios a simulação se pareça com um tecido, a nível dinâmico. Esta segunda abordagem, a da simulação baseada na resolução de sistemas de integração numérica, é então a que adquire aqui maior importância. Neste âmbito, é de referir que pode encontrar-se uma grande variedade de modelos físicos na literatura e, consequentemente, uma grande variedade de técnicas de integração, fenómenos que se podem explicar pela ausência de um modelo que seja bom em todos os aspectos.

De uma forma geral, na simulação baseada na resolução de sistemas de integração numérica, são calculadas as resultantes das forças, ou energias, que actuam sobre cada partícula do sistema, sendo que a forma como estas são calculadas varia consoante a técnica usada. É relevante referir que, no geral, a força resultante é resultado da actuação de dois grupos distintos de forças – forças internas e forças externas. As forças internas de um tecido

resultam das suas propriedades de deformação (Figura 3). Estas propriedades descrevem interacções entre partículas através das quais as forças são modeladas. A forma como as interacções são definidas varia consoante a técnica de modelação. As forças externas são forças como gravidade, vento, etc.

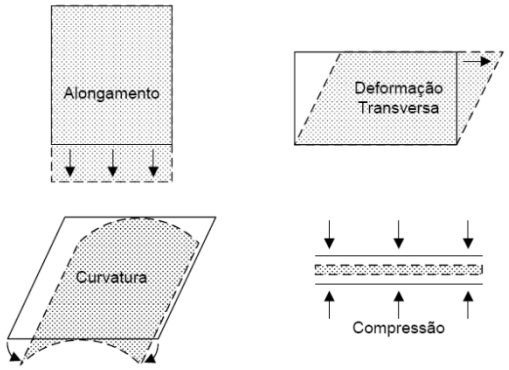

Figura 3 - Principais deformações do tecido, retirado de [Birr07].

No entanto, aqui não interessa abordar toda a teoria associada a cada modelo de tecidos. O que nos interessa, de facto, é abordar o que estes modelos envolvem a nível da determinação do sistema de equações lineares, do calculo das forças e do respectivo método de integração. Para uma leitura alargada sobre modelos de tecidos, aconselha-se a leitura de [Birr07] onde é feita uma descrição extensa sobre o assunto.

2.2 Simulação de tecidos através de integração numérica

O que existe de comum entre as várias técnicas baseadas em sistemas de partículas é o facto de nelas serem descritas as alterações no sistema recorrendo a equações diferenciais, as quais têm que ser integradas numéricamente, de forma explícita ou implícita, para cada partícula e em cada iteração da simulação.

O conjunto de métodos de integração para sistemas de equações diferenciais, encontra-se divido entre dois grupos fundamentais:

- Os métodos de integração explícita
- Os métodos de integração implícita

A diferença fundamental entre os métodos explícitos e os métodos implícitos reside no facto de que os métodos explícitos apenas se baseiam nas condições existentes no início do passo de integração para avançar no tempo, enquanto os métodos implícitos são expressos em termos das condições existentes no final do mesmo, tentando descobrir o próximo estado do sistema através da procura de um estado a partir do qual consigam recuar para o estado presente, retirando-lhe o passo de tempo.

De seguida, apresentam-se algumas propostas para a simulação de tecidos através de integração numérica que recorrem a métodos explícitos, apresentando algumas das limitações ou problemas que estes métodos apresentam. Apresenta-se depois a proposta de Baraff e Witkin [Bar98], que recorre a um método de integração implícita como forma de contornar alguns dos problemas encontrados nos métodos explícitos.

Por fim, apresentam-se algumas propostas para a aceleração da simulação de tecidos com recurso à programação do GPU.

2.3 Proposta de Provot

O método de Euler resulta directamente da expansão em série de Taylor de uma função diferenciável em torno de um ponto. Sendo $f(t)$ uma função real de uma variável real t. Então, o seu valor para uma vizinhança Δt de um ponto t_0 pode ser dado por:

$$f(t_0 + \Delta t) = f(t_0) + \Delta t f'(t_0) + \Delta t^2 \frac{f''(t_0)}{2!} + \Delta t^3 \frac{f'''(t_0)}{3!} + \cdots + \Delta t^n \frac{f^{(n)}(t_0)}{n!} + \cdots \qquad (2.1)$$

No caso específico do método de integração explícita de Euler, apenas são utilizados os dois primeiros termos da equação (2.1), resultando:

$$f(t_0 + \Delta t) = f(t_0) + \Delta t f'(t_0) \qquad (2.2)$$

A equação (2.2), quando aplicada sucessivamente, permite integrar, por aproximação, o valor de uma qualquer função derivável, para valores sucessivos do seu parâmetro, partindo de um valor inicial e de uma forma de avaliar a sua primeira derivada.

O método de integração explícita de Euler surge pela primeira vez aplicado à modelação de tecidos no modelo apresentado por Provot [Pro95], onde a evolução do sistema é governada pela lei fundamental da dinâmica newtoniana:

$$\boldsymbol{f_i} = m_i \boldsymbol{a_i} \qquad (2.3)$$

onde $\boldsymbol{f_i}$ representa a resultante das forças exercidas sobre a partícula i, de massa m_i, e $\boldsymbol{a_i}$ a aceleração provocada pela força $\boldsymbol{f_i}$.

A equação (2.3) é uma equação diferencial de segunda ordem, podendo ser reescrita da seguinte forma:

$$\frac{\partial^2 \boldsymbol{p_i}(t)}{\partial t^2} = \frac{\boldsymbol{f_i}(t)}{m_i} \qquad (2.4)$$

onde $\boldsymbol{p_i}(t)$ representa a função posição da partícula i no instante t. Este tipo de problema pode reduzir-se a um conjunto de equações diferenciais de primeira ordem por introdução de uma variável auxiliar, a velocidade $\boldsymbol{v}$:

$$\begin{cases} \dfrac{\partial \boldsymbol{v_i}(t)}{\partial t} = \dfrac{\boldsymbol{f_i}(t)}{m_i} \\ \dfrac{\partial \boldsymbol{p_i}(t)}{\partial t} = \boldsymbol{v_i}(t) \end{cases} \qquad (2.5)$$

O sistema obtido pode agora ser facilmente integrado ao longo do tempo recorrendo ao método de Euler explícito da equação (2.2), sendo obtidas as seguintes fórmulas de actualização do estado do sistema em função do tempo:

$$\begin{cases} \boldsymbol{v_i}(t_0 + \Delta t) = \boldsymbol{v_i}(t_0) + \Delta t \dfrac{\boldsymbol{f_i}(t_0)}{m_i} \\ \boldsymbol{p_i}(t_0 + \Delta t) = \boldsymbol{p_i}(t_0) + \Delta t \boldsymbol{v_i}(t_0) \end{cases} \qquad (2.6)$$

Como se pode ver, o método de Euler explícito é um método de integração simples que apenas requer o conhecimento das forças, velocidades e posições actuais de cada partícula. É o método de integração numérica mais simples e deve utilizar passos de tempo pequenos, pois trata-se de uma aproximação pouco precisa onde, quanto maior for o passo, maior será o erro associado. Existem formas de reduzir o erro associado ao método de Euler além da redução do passo de tempo, como o método *Midpoint* ou o método *Runge-Kutta* [Wit95]. Para mais pormenores sobre o modelo de tecidos proposto por Provot, recomenda-se a leitura de [Pro95].

2.4 Proposta de Zeller

Um outro método explícito de integração é o método de integração explícita de Verlet. Este método fornece maior estabilidade que o método explícito de Euler, além de garantir outras propriedades importantes para alguns sistemas físicos como, por exemplo, a preservação de área [Zel05]. Esta forma de integração numérica reduz o erro introduzido, calculando a nova posição de uma partícula i em função da sua posição no passo de tempo anterior, da sua posição actual e das forças que actuam sobre ela, não recorrendo a velocidades.

Em 2005, Zeller [Zel05] implementa o método de verlet para um simulador de tecidos, recorrendo a um passo de simulação Δt constante. A nova posição de cada partícula i é calculada a partir da anterior em função da equação

$$\boldsymbol{p_i}(t_0 + \Delta t) = \boldsymbol{p_i}(t_0) + k[\boldsymbol{p_i}(t_0) - \boldsymbol{p_i}(t_0 - \Delta t)] + \Delta t^2 \frac{\boldsymbol{f_i}(t_0)}{m_i} \qquad (2.7)$$

onde $\boldsymbol{p_i}(t)$ representa a posição da partícula i no momento t, $\boldsymbol{f_i}(t)$ representa a resultante das forças aplicadas à partícula no momento t, m_i a massa da partícula e k um coeficiente de amortecimento, normalmente próximo de 1. Esta equação (2.7) é obtida por aproximação de $\boldsymbol{p_i}(t_0 + \Delta t)$ e $\boldsymbol{p_i}(t_0 - \Delta t)$ pelos primeiros termos das suas expansões em série de Taylor, somando as equações obtidas e tendo em conta que $\boldsymbol{p_i}''(t_0) = m_i \boldsymbol{f}_i(t_0)$ [Zel05].

Tal como no modelo apresentado por Provot [Pro95], o sistema de partículas proposto por Zeller [Zel05] é também um sistema de massas e molas. Neste modelo, existem dois tipos de molas: molas estruturais e molas de tensão (Figura 4).

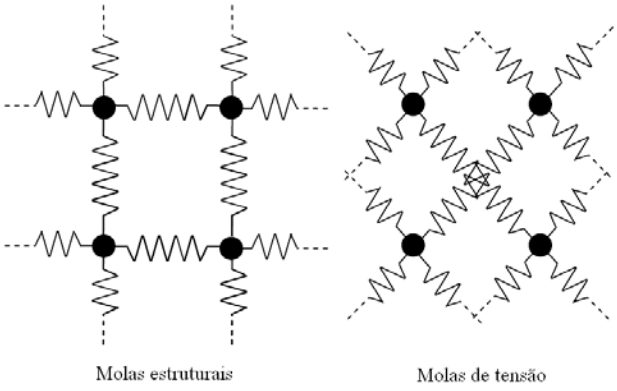

Figura 4 - Esquema de massas e molas proposto por Zeller [Zel05].

Considera-se que as molas têm rigidez infinita, ou seja, para duas partículas P_1 e P_2 unidas por uma mola, elas estão restritas tal que:

$$Dist(P_1, P_2) = d$$

onde d é uma constante igual ao tamanho da mola em repouso.

No caso de um pedaço de tecido rectangular liso, de tamanho (e_x, e_y), modelado por um vector de $(W \times H)$ partículas a distância d de cada mola é calculada da seguinte forma:

$$d_{estrutural,\ x} = \frac{e_x}{(W-1)}$$

$$d_{estrutural,\ y} = \frac{e_y}{(H-1)}$$

$$d_{tensão} = \sqrt{(d^2{}_{estrutural,\ x} + d^2{}_{estrutural,\ y})}$$

No caso de um pedaço de tecido não-rectangular ou não liso, cada mola tem uma distância d específica em repouso.

Apesar da simplicidade e eficácia dos modelos apresentados, os métodos explícitos apresentam algumas desvantagens. O maior problema, como descrito em [Bar01], é que estes não lidam bem com forças de grande rigidez, como as forças internas do tecido. Este problema deriva do facto dos métodos explícitos serem lineares no que toca ao tamanho do passo de tempo [Bar01]. Se o passo de tempo Δt for aumentado, a instabilidade do estado do sistema ao longo das iterações aumenta também. Um bom exemplo é se tentarmos aproximar uma equação que descreva um movimento circular infinito através do método explícito de Euler (Figura 5). O erro associado a esta aproximação não lhe permite desenhar este movimento correctamente, sendo que a aproximação conseguida é uma espiral. O aumento do passo apenas define a «largura» da espiral, sendo que quanto maior for o passo de integração, mais «larga» será a espiral, aumentando drásticamente o erro da aproximação a cada iteração.

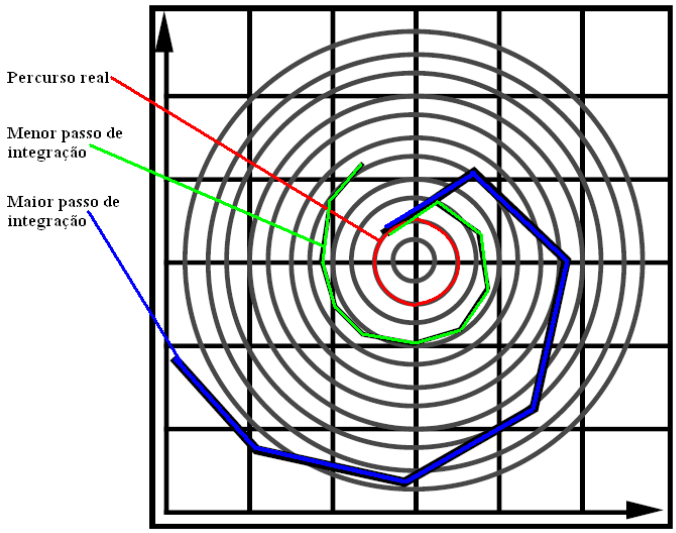

Figura 5 - Erro associado ao método explícito de Euler.

Existe um conjunto de abordagens para evitar as oscilações dos métodos explícitos, como aplicar forças de amortecimento, fazer pós-processamento por restrições, ou reduzir o passo de simulação. O que produz melhores resultados na prática, passa por esta última e pela aplicação de restrições após a resolução do sistema linear, pois na prática nunca é suficiente aplicar apenas uma das alternativas. Em [Wit95], sugere-se também a utilização de métodos como o método *Midpoint* ou o método de *Runge-Kutta* como uma boa forma de reduzir o erro associado a métodos explícitos, se bem que qualquer um dos métodos implica cálculos de maior complexidade. Reduzindo o passo de simulação, o sistema de equações terá de ser resolvido mais vezes por cada *frame*, reduzindo o *frame-rate* e, portanto, degradando a performance da simulação. Também a aplicação posterior de restrições implica mais processamento de elevado custo pelo que também contribui para uma degradação da performance da simulação.

Uma solução que permite usar um passo de intergração maior, sem a necessidade da aplicação de forças de amortecimento ou de pós-processamento por restrições, por não possuir o mesmo erro para passos de integração maior, é a utilização de um método de integração implícita, como se apresenta seguidamente.

2.5 Proposta de Baraff e Witkin

O método de integração implícita de Euler tenta descobrir o próximo estado do sistema, procurando um estado a partir do qual consiga recuar para o estado presente, retirando-lhe o passo de tempo. O método implícito de Euler (também conhecido como *Backwards Euler Method*) resulta então no seguinte sistema de equações, reproduzido na forma vectorial:

$$\begin{pmatrix} \Delta \boldsymbol{v} \\ \Delta \boldsymbol{p} \end{pmatrix} = \Delta t \begin{pmatrix} \boldsymbol{M^{-1}} \boldsymbol{f}(\boldsymbol{p_0} + \Delta \boldsymbol{p},\ \boldsymbol{v_0} + \Delta \boldsymbol{v}) \\ \boldsymbol{v_0} + \Delta \boldsymbol{v} \end{pmatrix} \qquad (2.8)$$

Aqui, o vector $\boldsymbol{v_0}$ contém as velocidades de cada partícula no início do passo de simulação, sendo que $\boldsymbol{p_0}$ representa o vector das posições de cada partícula no mesmo instante. Os valores $\Delta \boldsymbol{v}$ e $\Delta \boldsymbol{p}$ representam as variações da velocidade e da posição do sistema para cada partícula, respectivamente. O passo de simulação é representado por Δt e $\boldsymbol{M^{-1}}$ é uma matriz diagonal que representa o inverso das massas de cada partícula, ou seja, para uma partícula i, $\boldsymbol{M_i}^{-\mathbf{1}} = \frac{1}{m_i}$, sendo m_i a massa da partícula i.

Conforme se verá adiante, é importante perceber que, apesar de apresentar uma melhor solução, esta equação leva a um conjunto esparso de nxn equações lineares que tem de ser resolvido a cada iteração, sendo n o número total de partículas. Por outro lado, como apontado em [Bar98], dada a natureza esparsa do método implícito e o facto de ser possível usar passos de integração maiores para cada iteração, esta complexidade acrescida em relação ao método explícito é largamente compensada.

Em 1998, David Baraff e Andrew Witkin [Bar98] apresentaram um trabalho que é considerado um marco no panorama da simulação de tecidos pela comunidade gráfica e que recorre a uma técnica de integração implícita. Neste modelo, o tecido é também modelado segundo uma malha triangular onde nos vértices dos triângulos se encontram pontos de massa discretizada – partículas. No entanto, apesar deste carácter discreto, o modelo que é usado para avaliar a dinâmica do tecido é um modelo contínuo, o que permite que se possa especificar um tecido independentemente do nível de discretização usado.

O modelo proposto em [Bar98] descreve uma peça de roupa como pedaços planos de peças de tecido, as quais tendem a resistir a alterações a este estado inicial plano. Para capturar este equilibrio inicial, a cada partícula i, cuja posição no espaço 3D é dada por $\boldsymbol{p}(i) = (x_i, y_i, z_i)$, é atribuído um par de coordenadas (u_i, v_i), que correspondem às coordenadas da partícula no plano do tecido. Estas coordenadas são usadas para, por exemplo, avaliar a tensão ou deformação transversa exercidas no tecido a cada instante, ou também para efectuar o mapeamento de texturas no tecido. Os autores consideram ainda que se deve imaginar, a cada instante, a existência de uma função contínua $w(u_i, v_i)$ que transforma as coordenadas 2D de cada partícula para a sua respectiva posição no espaço 3D (Figura 6) bem como a existência das respectivas derivadas parciais, $\boldsymbol{w}_u$ e $\boldsymbol{w}_v$, definidas por:

$$\boldsymbol{w_u} = \frac{\partial \boldsymbol{w}}{\partial u} \quad e \quad \boldsymbol{w_v} = \frac{\partial \boldsymbol{w}}{\partial v}$$

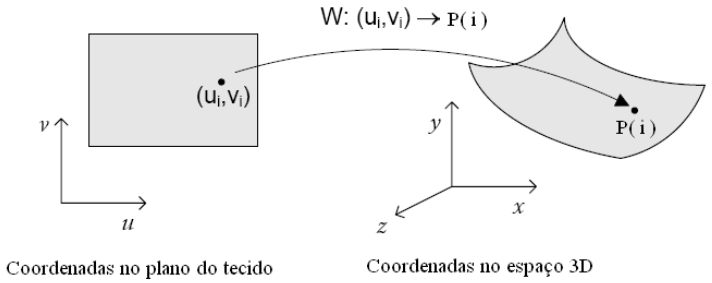

Figura 6 - Função *w* segundo o modelo de Baraff e Witkin.

Ao contrário do que acontece nos modelos explícitos apresentados por Zeller [13, 14] ou por Provot [Pro95], onde são definidas expressões directas para a energia associada a cada uma das forças internas, no modelo proposto por Baraff e Witkin, em [Bar98], opta-se pela utilização de vectores condição como forma de definir estas energias.

Um vector condição é representado por $\boldsymbol{C}(\boldsymbol{x})$ e é formulado de forma a ser nulo no estado de equilíbrio, sendo que $\boldsymbol{x}$ representa o vector de posições de todo o sistema, ou seja, o vector formado por todas as posições $\boldsymbol{p}(i)$ das partículas do sistema. São avaliados três tipos de forças internas, provocados pelos fenómenos de deformação transversa, de curvatura e de tensão. As forças relacionadas com deformação transversa e de tensão, são avaliadas individualmente, triângulo a triângulo. Já a força de curvatura é avaliada a cada par de triângulos adjacentes. No caso da tensão interna do tecido, a condição proposta é a de medir o módulo das derivadas parciais da função $\boldsymbol{w}(u, v)$ ao longo de cada uma das direcções principais. Fazendo coincidir o sistema de coordenadas uv com as direcções principais do tecido, pode medir-se a tensão, de forma independente, nas direcções da teia e da trama. De facto, estas derivadas parciais medem efectivamente o alongamento ou a compressão do tecido, segundo estas direcções principais. Se os valores dos módulos das suas derivadas

parciais, $\|\boldsymbol{w}_u\|$ e $\|\boldsymbol{w}_v\|$, forem unitários o tecido não se encontra sobre tensão, não estando alongado ou comprimido segundo nenhuma direcção.

O vector condição para a tensão interna do tecido proposto por Baraff e Witkin é então definido por:

$$C_{tensão}(\boldsymbol{x}) = a_\Delta \begin{pmatrix} \|\boldsymbol{w_u}(\boldsymbol{x})\| - b_u \\ \|\boldsymbol{w_v}(\boldsymbol{x})\| - b_v \end{pmatrix} \qquad (2.9)$$

O valor de a_Δ representa a área total do triângulo no sistema de coordenadas fixo uv e os valores de b_u e b_v são, geralmente, de valor igual a um. É de referir que b_u e b_v são parâmetros que permitem modelar de uma forma simples diversos aspectos dos tecidos. Por exemplo, se considerarmos uma manga de uma camisola, sendo u a direcção do ombro para a mão e v a direcção perpendicular (que no sistema de coordenadas 3D descreve a curvatura do «cilindro» da manga), podemos caracterizar o tecido para se parecer mais com uma manga, tornando-o mais flexível na direcção do braço e menos em pontos de aperto como o pulso, simplesmente alterando os valores de b_u e b_v. Para a «folga» na direcção de u, especifica-se um valor de b_u superior à unidade. Para se apertar o tecido na zona junto ao pulso, basta atribuir localmente um valor de b_v inferior à unidade.

A condição proposta em [Bar98] para a deformação transversa é obtida avaliando o ângulo β apresentado na figura (Figura 7). Este ângulo pode ser obtido simplesmente através do produto interno das derivadas parciais $\boldsymbol{w_u}$ e $\boldsymbol{w_v}$. Tendo em conta que as condições impostas às forças de tensão não permitem que o tecido se alongue demasiado, os módulos de $\boldsymbol{w_u}$ e $\boldsymbol{w_v}$ podem ser considerados unitários. Então, a condição para a deformação transversa é definida simplesmente por:

$$C_{deformação\ transversa}(\boldsymbol{x}) = a_\Delta \boldsymbol{w_u}(\boldsymbol{x})^T \boldsymbol{w_v}(\boldsymbol{x}) \qquad (2.10)$$

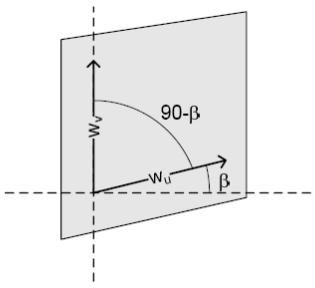

Figura 7 - avaliação da deformação transversa.

A condição para a curvatura consiste em fazer a avaliação do ângulo θ entre duas faces triangulares, na aresta partilhada entre ambas. Este valor pode ser calculado com base em relações trignométricas entre as normais a cada um dos triângulos. Dado que o calculo se baseia num par de triângulos adjacentes, com uma aresta partilhada, a condição depende de quatro partículas, em vez das três necessárias às condições de tensão e deformação transversa.

Como referido antes, estas condições servem para fazer a avaliação das forças internas ao plano do tecido. Em [Bar98], propõe-se que a energia respectiva a um vector de condição $C(\boldsymbol{x})$ seja obtida a partir da expressão:

$$E\big(C(\boldsymbol{x})\big) = \frac{k}{2} C(\boldsymbol{x})^T C(\boldsymbol{x}) \qquad (2.11)$$

Nesta expressão, k é um coeficiente de rigidez. A força provocada por esta energia potencial é dada por $\boldsymbol{f} = -{}^{\partial E}/_{\partial \boldsymbol{x}}$. Assim, a força aplicada a cada partícula i envolvida em $C(\boldsymbol{x})$, é dada por:

$$\boldsymbol{f_i} = -k \frac{\partial C(\boldsymbol{x})}{\partial \boldsymbol{x}_i} C(\boldsymbol{x}) \qquad (2.12)$$

Esta equação depende de um conjunto muito reduzido de partículas e dá origem a um vector total de forças, $\boldsymbol{f}$, muito esparso. Ao recorrer à formulação destes vectores condição, torna-se mais fácil introduzir forças de amortecimento sensíveis apenas aos fenómenos internos do tecido.

Existem vários aspectos importantes para a eficiência computacional desta técnica. Em primeiro lugar, o facto de ser usado o método implícito, cujas vantagens já foram apresentadas. Além disso, apesar de já ter sido abordada a integração implícita, no modelo contínuo proposto por Terzopoulos em [Ter87] onde a integração do movimento é feita recorrendo à equação de Lagrange, o método proposto em [Bar98] traz a novidade de permitir a satisfação de restrições, existentes sobre as partículas em simultâneo com a integração, sem aumentar a complexidade temporal. Pode-se ainda referir o aumento da estabilidade da simulação, o que permite a utilização de forças com rigidez maior pois, ao contrário do que acontece com os modelos explícitos onde uma maior rigidez das forças internas do tecido obriga a uma redução do passo de tempo para manter a estabilidade, no método implícito o passo de simulação é praticamente insensível à rigidez destas forças.

Relembremos, então, o sistema de equações (2.8):

$$\begin{pmatrix} \Delta \boldsymbol{v} \\ \Delta \boldsymbol{p} \end{pmatrix} = \Delta t \begin{pmatrix} \boldsymbol{M}^{-1} \boldsymbol{f}(\boldsymbol{p_0} + \Delta \boldsymbol{p},\ \boldsymbol{v_0} + \Delta \boldsymbol{v}) \\ \boldsymbol{v_0} + \Delta \boldsymbol{v} \end{pmatrix}$$

A nível operacional, existe uma grande diferença entre este método e o explícito. Para cada passo de simulação Δt, o método explícito apenas requer uma avaliação da função $\boldsymbol{f}$, que calcula as forças sobre todas as partículas. Já o método implícito necessita que o sistema de equações (2.8) seja resolvido em ordem às incógnitas $\Delta \boldsymbol{v}$ e $\Delta \boldsymbol{p}$. Trata-se de um sistema não linear e a solução mais comum para o aproximar passa pela utilização da expansão em série de Taylor. Para o método implícito de Euler de primeira ordem, a expansão é feita apenas até às primeiras derivadas, resultando então:

$$\boldsymbol{f}(\boldsymbol{p_o} + \Delta \boldsymbol{p}, \boldsymbol{v_0} + \Delta \boldsymbol{v}) = \boldsymbol{f_0} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \Delta \boldsymbol{p} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} \Delta \boldsymbol{v} \qquad (2.13)$$

onde $\boldsymbol{f_0} = \boldsymbol{f}(\boldsymbol{p_0}, \boldsymbol{v_0})$ e as derivadas parciais são avaliadas para este estado inicial (p_0, v_0). Usando esta aproximação, e usando a segunda linha de (2.8) para eliminar a variável $\Delta \boldsymbol{p}$ na primeira parcela do mesmo sistema, obtém-se a seguinte equação:

$$\Delta \boldsymbol{v} = \Delta t \boldsymbol{M}^{-1} \left(\boldsymbol{f_0} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \Delta t (\boldsymbol{v_o} + \Delta \boldsymbol{v}) + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} \Delta \boldsymbol{v} \right) \qquad (2.14)$$

Se isolarmos $\Delta \boldsymbol{v}$, obtém-se

$$\left(\boldsymbol{I} - \Delta t \boldsymbol{M}^{-1} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} - \Delta t^2 \boldsymbol{M}^{-1} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \right) \Delta \boldsymbol{v} = \Delta t \boldsymbol{M}^{-1} \left(\boldsymbol{f_0} + \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \boldsymbol{v_0} \right) \qquad (2.15)$$

Esta equação apresenta no entanto um problema. A menos que todas as partículas no sistema possuam a mesma massa, a matriz do sistema não é simétrica. Para tornar a matriz do sistema simétrica e definida-positiva, basta multiplicar tudo pela matriz M, obtendo-se o sistema:

$$\left(\boldsymbol{M} - \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} - \Delta t^2 \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \right) \Delta \boldsymbol{v} = \Delta t \left(\boldsymbol{f_0} + \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \boldsymbol{v_0} \right) \qquad (2.16)$$

Trata-se de um sistema linear, do tipo $Ax = b$, que pode agora ser resolvido por métodos iterativos como o método dos gradientes conjugados [She94], o qual permite obter uma boa solução em poucas iterações, sem modificar a matriz do sistema. A operação mais complexa a cada iteração é, assim, a multiplicação desta matriz por um vector. Este facto, associado à ausência de alteração da matriz e às propriedades esparsas da mesma, torna o método dos gradientes conjugados atraente para situações onde a matriz seja de grandes dimensões.

Recomenda-se o aprofundamento do conhecimento sobre o modelo de tecidos proposto por Baraff e Witkin, através da leitura de [Bar98] e de [Birr07] e, embora não nos interesse aqui abordar outro método iterativo para a resolução deste sistema linear, é de referir que existem alternativas ao método dos gradientes conjugados para a resolução do sistema, como o método de Jacobi por exemplo. Para uma percepção mais aprofundada sobre este tipo de métodos iterativos, aconselha-se a leitura de [She94].

2.6 Implementações em GPU

Como podemos ver, a resolução de sistemas de equações diferenciais para um sistema de partículas é uma tarefa computacionalmente muito intensiva. Afinal, no caso dos modelos explícitos tem de se executar um mesmo cálculo para todas as partículas do sistema e, no caso dos modelos implícitos, há um sistema linear de equações para resolver.

Dado o peso computacional destas operações, torna-se interessante a sua paralelização, por forma a acelerar o cálculo. É de referir também que as operações realizadas envolvem matrizes e vectores e, sendo o GPU uma unidade de processamento paralelo optimizado para operações sobre matrizes e vectores, torna-se atraente a ideia de executar todo este processamento no GPU.

2.6.1 Implementação de Zeller

Conforme se viu anteriormente, Zeller [Zel05] utilizou o método de integração de Verlet para implementar um simulador de tecidos no GPU para animação em tempo real (Figura 8). No modelo que propõe, as texturas servem para guardar as posições das partículas. A cada instante, as novas posições são calculadas através de vários *pixel shaders*, sendo gravadas noutra textura. Um *vertex shader* é utilizado, posteriormente, para fazer a síntese do tecido a partir das novas posições obtidas.

Figura 8 - Simulador de tecidos por Zeller [Zel05], NVIDIA SDK 9.

O algoritmo proposto em [Zel05] baseia-se em três passos. O primeiro passo consiste em aplicar o método de integração de Verlet. Este passo é de implementação simples e directa em GPU, dado que o cálculo é igual e independente para cada partícula, logo facilmente paralelizável. O segundo passo deve satisfazer as restrições de distância entre as molas (Figura 4) para evitar fenómenos de super-elasticidade como os descritos por Provot em [Pro95]. Este passo já não é tão trivial de paralelizar dado que existe mais do que uma mola a actuar sobre cada partícula. Para lidar com este problema, Zeller propõe em [Zel05] a utilização de um conjunto de 8 *pixel shaders* em que cada um lida com grupos de linhas ou colunas separadas de molas (sem interacção entre elas). Cada um destes *pixel shaders* escreve os resultados para uma textura diferente. Num, são satisfeitas as restrições das molas estruturais, segundo o eixo dos x, das colunas pares, noutro das molas estruturais, também segundo o eixo dos x, das colunas impares. Segundo o eixo dos y, existem também dois *pixel shaders*, das molas estruturais das linhas pares e das molas estruturais das linhas impares. O mesmo esquema se aplica depois às molas de tensão (Figura 9).

Ao definir estes grupos individuais e independentes, os cálculos tornam-se trivialmente paralelizáveis podendo ser executados em qualquer ordem e, portanto, eficientemente executados em GPU. Para mais pormenores sobre a implementação que Zeller propõe, nomeadamente a forma como são tratadas as colisões e como é feita a síntese final do tecido, remete-se a leitura de [Zel05], dado que estes detalhes estão já fora do âmbito deste trabalho.

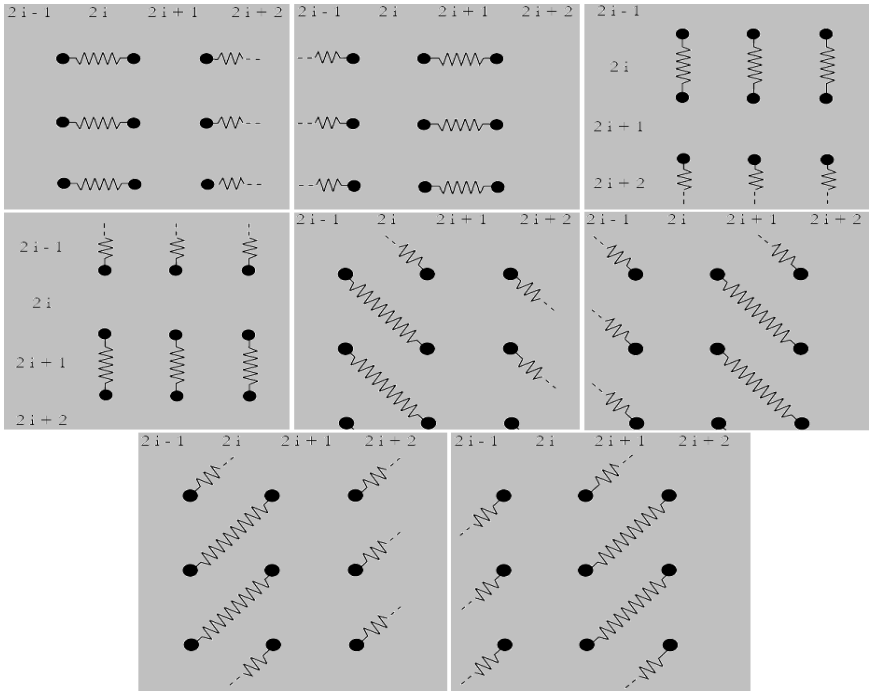

Figura 9 - Os oito esquemas de molas independentes processados nos *pixel shaders* para imposições de restrições em [Zel05].

Com a evolução do GPU e com o surgimento de um novo tipo de *shaders*, o *geometry shader*, Zeller apresenta em [Zel07] uma nova versão do seu simulador (Figura 10).

Figura 10 - Simulador de tecidos proposto por Zeller em [Zel07], NVIDIA SDK 10.

A nova proposta de Zeller é diferente da anterior apenas em alguns pormenores da implementação em GPU. As partículas são guardadas num *vertex buffer* e o trabalho é feito sobretudo durante os passos de processamento de vértices (*vertex shading*) e de geometria (*geometry shading*), enquanto que na versão proposta em [Zel05] as partículas eram guardadas em texturas e a maior parte do processamento se dava na fase de processamento de pixeis (*pixel shading*). Uma das novidades que o *geometry shader* trouxe, é que um passo neste *shader* pode retornar vários vértices, o que permite lidar com mais do que uma partícula em apenas uma chamada do *shader*, tornando a implementação mais eficiente. Permite assim uma redução do número de passos e torna possível que as restrições de distância entre cada par de partículas seja computada em apenas quatro passos, em vez dos oito apresentados em [Zel05].

As restrições sobre as molas envolvem sempre dois vértices (partículas). Dado que dois vértices permitem definir uma linha, podemos descrever uma mola como uma linha, logo estas restrições poderão ser processadas na fase de *geometry shading* como linhas (uma linha é uma primitiva geométrica, logo processável na fase de geometria). O *geometry shader* permite até 6 vértices de entrada. Como referido em [Zel07], a escolha do número óptimo de vértices de entrada não é uma escolha intuitiva. Por um lado, se existirem menos vértices de entrada na chamada do *shader*, serão necessários mais passos de síntese, mas por outro lado, a performance do processamento de geometria degrada-se quantos mais vértices forem utilizados. Assim, Zeller escolheu o processamento de quatro vértices por cada chamada, justificando a escolha como a melhor solução na prática.

Para o processamento ser efectuado em paralelo no GPU, é necessário que as restrições aplicadas em cada passagem sejam independentes entre si, seja por representarem tipos diferentes de molas, ou por não haver partículas partilhadas entre cada grupo de quatro vértices a processar. Assim, Zeller propõe a divisão do processamento de restrições em quatro passagens por grupos de molas independentes (Figura 11).

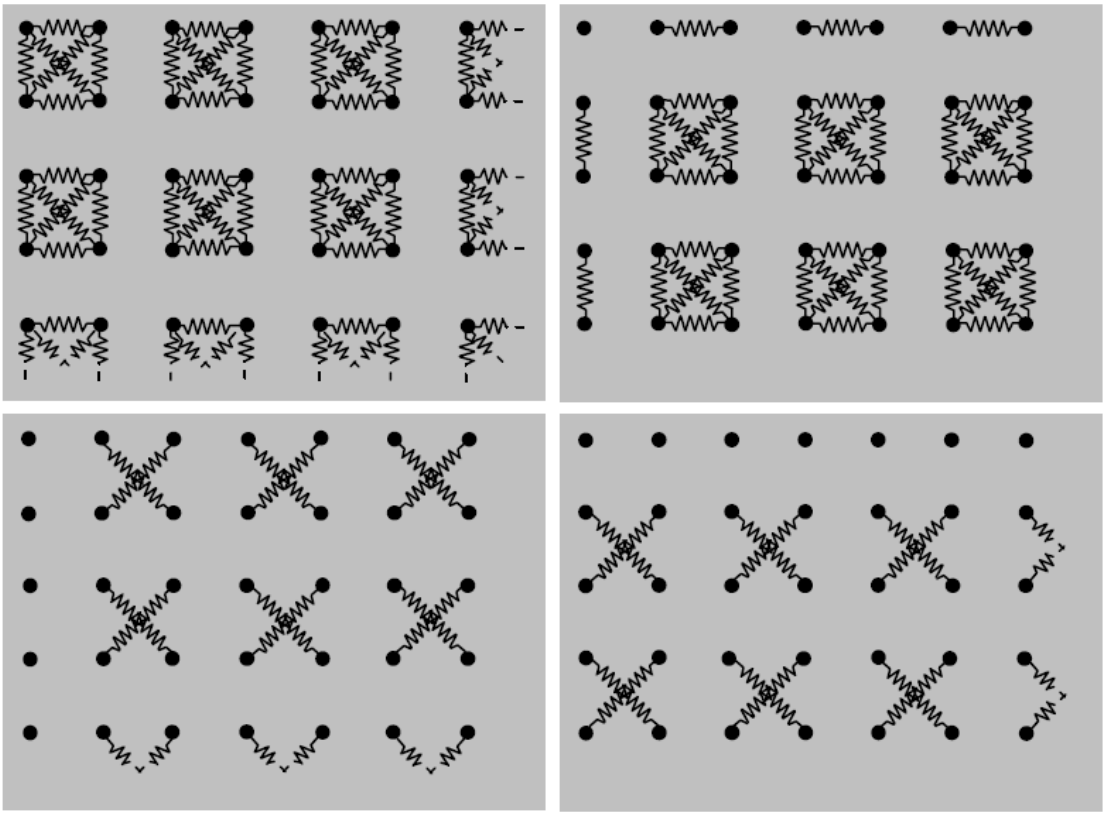

Figura 11 - Particionamento do conjunto total de restrições em quatro conjuntos independentes de molas, para avaliação paralela no GPU. Proposto por Zeller em [Zel07].

2.6.2 Implementação de Kjartan Dencker

Em 2006, Kjartan Dencker desenvolveu, para a sua dissertação de mestrado [Den06], um simulador de tecidos em GPU segundo um modelo de integração implícito. Em [Den06], o objectivo principal foi comparar o desempenho do simulador de tecidos implementado em GPU com uma mesma implementação em CPU. A maior contribuição foi que, para este estudo, o autor se inspirou no modelo implícito de Baraff e Witkin [Bar98] e explorou a utilização do método dos gradientes conjugados, bem como do método de Jacobi, no GPU. O autor recorreu a programação genérica sobre o GPU, através da utilização de uma API gráfica, tal como Zeller.

No entanto, dados os objectivos principais do projecto, em [Den06] foram feitas simplificações ao modelo. Em vez de uma malha de triângulos, usou-se uma malha de quadriláteros, sendo as partículas do sistema representadas com recurso a duas texturas, uma para as posições e outra para as velocidades. Além disto, o modelo de massas e molas é diferente do utilizado em [Bar98], estando cada partícula ligada por molas às 24 partículas mais próximas. Na figura (Figura 12) mostra-se um exemplo de uma textura para posições e como seria a correspondente malha. Na figura (Figura 13) mostra-se uma malha com 25 partículas e o esquema de massas e molas respeitante à partícula central da malha, mostrando as 24 ligações referidas.

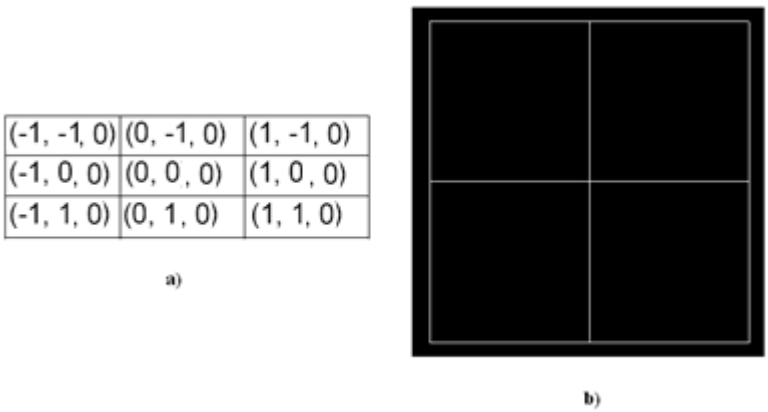

Figura 12 - a) Exemplo de uma textura com posições. Cada entrada tem coordenadas xyz. b) Malha representada pela textura em a).

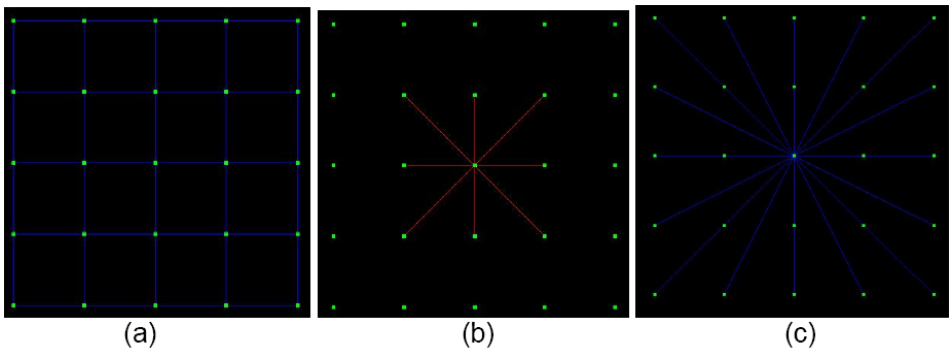

Figura 13 - a) Malha com 25 partículas. b) Molas de tensão e alongamento da partícula central. c) Molas de curvatura da partícula central.

Para a implementação do método implícito foi então necessário implementar três estruturas distintas a nível do GPU, matrizes esparsas, matrizes diagonais e vectores.

Os vectores foram implementados como texturas rectangulares onde cada pixel corresponde a uma partícula. A velocidade de uma partícula tem a mesma coordenada na textura de velocidades que a posição dessa mesma partícula possui na textura de posições. A textura utilizada foi uma textura 2D, sendo feito mapeamento directo dos índices de vector para coordenadas de textura. Uma índice de vector, i, de uma partícula é transformado nas coordenadas de textura $x = i\%width$ e $y = \frac{i}{width}$ usando divisão inteira, onde $width$ é a «largura» da textura (Figura 14).

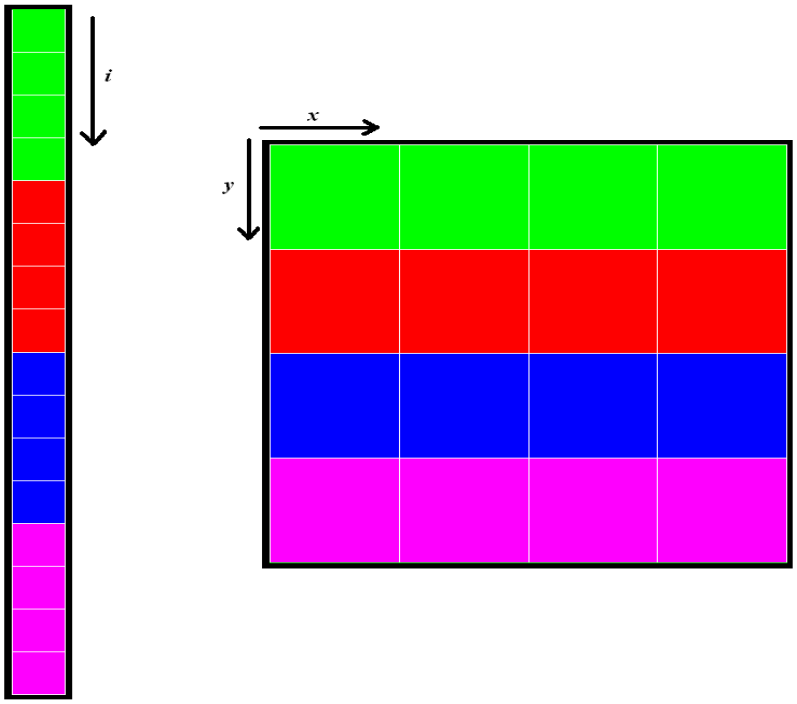

Figura 14 - Transformação dos índices de vector, à esquerda, em coordenadas de textura, à direita.

As matrizes diagonais foram construídas de forma semelhante. Primeiro, a cada posição da diagonal da matriz $M_{i,i}$ é atribuído o respectivo índice i de um vector e este é depois mapeado para coordenadas de textura da mesma forma que qualquer outro vector.

É de referir que cada elemento numa matriz é uma matriz de 3x3, o que significa que se têm de guardar nove valores para cada pixel. Para tal, as matrizes são representadas com três texturas, sendo que na primeira textura são representados os valores da primeira linha das matrizes 3x3, na segunda textura os da segunda linha das matrizes 3x3 e a terceira textura com as terceiras linhas das matrizes 3x3. Assim, sobre cada pixel são guardados os nove valores, sendo a informação distribuída num mesmo pixel, em três texturas diferentes, três valores em cada.

As matrizes das derivadas parciais são matrizes esparsas e contêm as derivadas entre quaisquer duas partículas, apresentando-se vazia em todos os pares (i, j) onde i não se relaciona com j. Uma vez que cada partícula i se relaciona com 24 outras partículas (Figura 13), torna-se necessária uma textura com apenas 25 colunas para a representação destas matrizes. Assim, é feito um mapeamento das coordenadas (i, j) da matriz esparsa original para coordenadas de textura (y, x), sendo x calculado de forma a estar contido neste espaço de 25 colunas e y usado directamente com o valor de i. Este modelo permite saber, em função desta coordenada x na textura, o valor da coluna j da matriz esparsa original.

Para a avaliação do modelo, são utilizados três *pixel shaders*, um para a determinação das forças, e outros dois para a determinação das derivadas parciais.

Uma vez feita a avaliação do modelo, são usados mais dois *pixel shaders* para determinar os valores de A e b do sistema de equações lineares a resolver, $Ax = b$.

Uma vez determinados A e b, pode-se então passar à resolução do sistema linear de equações. Tal como referido antes, em [Den06] foram implementados dois métodos numéricos para a resolução do sistema linear de equações, o método dos gradientes conjugados e o método de Jacobi.

Para o método dos gradientes conjugados, são necessárias várias passagens de síntese, envolvendo multiplicação de matrizes esparsas por vectores, multiplicação de matrizes diagonais por vectores, produto interno entre vectores e soma e subtração de vectores.

As operações de multiplicação de matrizes diagonais por vectores, e de soma e subtracção de vectores, são implementadas de forma simples, uma vez que a operação para cada pixel usa dados da mesma posição nas outras duas texturas.

Já a operação de multiplicação entre matrizes esparsas e vectores é um pouco mais complicada. O resultado é um vector e, no *pixel shader*, recebe-se a posição i para a qual se deve escrever. Em cada pixel calcula-se, então, a multiplicação da linha i da matriz esparsa por todo o vector. Para cada linha i da matriz esparsa é necessário calcular qual o índice j onde se encontra o vértice 3x1 do vector pelo qual cada matriz 3x3 de posição (y, x) da matriz esparsa deve ser multiplicado.

O produto interno entre vectores é implementado de forma mais simples. Primeiro, é feito um produto interno entre pixeis correspondentes, sendo o resultado guardado numa textura intermédia. É depois necessário fazer uma redução até obter uma textura 1x1 representando o valor. Cada operação num pixel soma até quatro valores do resultado anterior.

No caso do método de Jacobi a implementação em GPU foi bastante mais simples do que para o método dos gradientes conjugados, sendo que todos os calculos que são necessários fazer a cada iteração podem ser reduzidos a um único passo de síntese através de um único *pixel shader*.

É interessante referir alguns dos resultados obtidos em [Den06]. A placa gráfica utilizada para o desenvolvimento foi uma NVIDIA GeForce 6800 Go, cujo GPU possui 12 processadores de pixeis e 5 processadores de vértices. Para se fazer a análise de performance foram tidas em conta duas métricas básicas:

- O número de iterações que se permite aos métodos numéricos usar para converger para o vector solução. Este número varia entre 10 e 70 iterações. O autor afirma que o uso de 30 iterações resulta numa boa simulação na prática.
- O número de partículas usadas na simulação. Este valor varia entre 100 e 3600 partículas, sendo 4096 o número máximo de partículas para a implementação em GPU, uma limitação derivada do tamanho máximo das texturas.

As implementações em CPU apresentam performance superior para números mais pequenos de partículas mas, quando este número começa a aumentar, aproximando-se das 1000 partículas, as implementações em GPU começam a apresentar performances bastante superiores (Figura 15) às de CPU.

Como é também vísivel na figura (Figura 15), o método de Jacobi apresenta performances ligeiramente superiores às do método dos gradientes conjugados. Este fenómeno pode ser explicado pelo facto de que a implementação do método dos gradientes conjugados implica várias passagens pela fase processamento de pixeis, ao contrário do método de Jacobi que necessita de apenas uma. No entanto, o autor refere que o método de Jacobi apresenta uma aproximação menos realista, além de se mostrar bastante mais instável que o método dos gradientes conjugados quando se usam passos de tempo maiores. Para mais pormenores sobre o trabalho de Kjarten Dencker, sugere-se a leitura de [Den06].

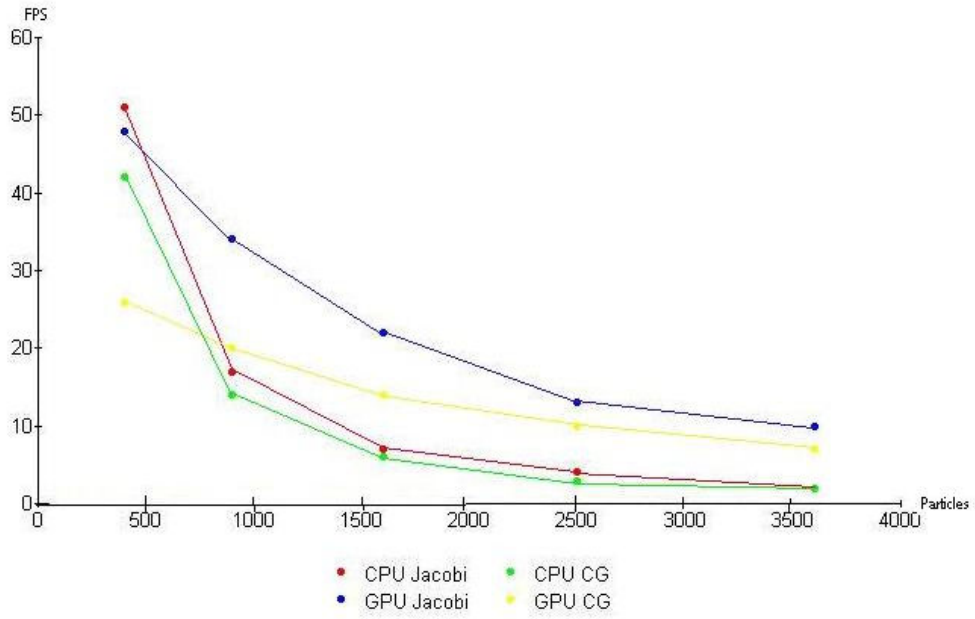

Figura 15 - esta figura, retirada de [Den06], apresenta como as diferentes implementações dos diferentes métodos se comportam a nível de performance. Todos os valores no gráfico utilizam 30 iterações para convergir.

Os ganhos em tempo de execução, para números maiores de partículas, nas implementações em GPU, contribuem para a motivação deste trabalho. Afinal, foram obtidas em [Den06] simulações interactivas (30fps) para números elevados de partículas, usando 30 iterações para convergência, usando o método dos gradientes conjugados. No entanto, o modelo proposto em [Den06] é um modelo muito rígido. A falta de flexibilidade da solução proposta é visível, por exemplo, na forma como são guardadas as matrizes esparsas, cuja estrutura é totalmente dependente do modelo de massas e molas usado. Além disso, existem limites no número máximo de partículas do sistema, causados por limites no tamanho das texturas, associados à programação do GPU através de APIs gráficas.

Para a fase de desenvolvimento desta dissertação, pretende-se utilizar uma NVIDIA GeForce 8800 GTS, que possui 96 processadores paralelos para processamento de pixeis, vértices ou geometria. Se tivermos em conta os 12 processadores de pixeis do GPU utilizados em [Den06], tornam-se interessantes as perspectivas de ganho esperado, em termos de tempo de execução. Além disto, prevê-se que a utilização do modelo de programação CUDA, em alternativa à utilização de linguagens de *shading*, permita utilizar estruturas de dados mais apropriadas para solucionar este tipo de problemas de uma forma mais genérica e flexível.

Assim, no capítulo seguinte apresenta-se a plataforma CUDA, abordando temas como o seu modelo de programação, o seu modelo de memória e o modelo de execução, sendo abordadas, ainda, algumas considerações sobre a arquitectura física do GPU.

No 4º Capítulo, é apresentado o método dos gradientes conjugados. Este algoritmo iterativo foi escolhido como alvo de paralelização por representar uma grande parte da computação realizada na fase *Solve*, que é identificada em [Birr07] como sendo a fase mais pesada computacionalmente (ver figura 2). São ainda apresentados alguns argumentos que justificam a escolha deste método para alvo de paralelização.

No 5º Capítulo são apresentados os estudos realizados com vista à implementação em GPU do método dos gradientes conjugados pre-condicionado modificado (MPCG), implementado em CPU no trabalho realizado por [Birr07] e apresentado no 4º Capítulo, sendo ainda apresentados alguns resultados respeitantes a esses mesmos estudos, como a comparação de performance entre diferentes implementações para cada operação necessária à realização em GPU do MPCG. É depois apresentada a abordagem tomada para a implementação do MPCG em GPU, inserido no trabalho realizado por [Birr07].

Finalmente, nos 6º e 7º Capítulos, são apresentados e discutidos os resultados obtidos e são apresentadas algumas sugestões para trabalho futuro.

3. NVIDIA CUDA

Até há pouco tempo, aceder ao poder computacional do GPU de forma eficiente era difícil pois o GPU apenas podia ser programado através de APIs gráficas, obrigando os programadores a tratar os problemas como problemas gráficos e apresentando algumas restrições, tais como o limite no tamanho de texturas. Além disto, os programas possíveis de realizar com APIs gráficas apenas podiam ler a RAM da placa gráfica de forma genérica, não podendo escrever também de forma genérica [Gov07], o que retirava muita da flexibilidade disponível na programação em CPU. O CUDA foi lançado pela NVIDIA como resposta a estas questões. Em CUDA, o GPU consiste num conjunto de multiprocessadores, de 8 *cores* cada, com arquitectura SIMD – *single instruction multiple data*. Cada um destes multiprocessadores é capaz de executar 32 vezes a mesma instrução sobre dados diferentes a cada 4 *clocks*, executando cada *core* 4 vezes a mesma instrução.

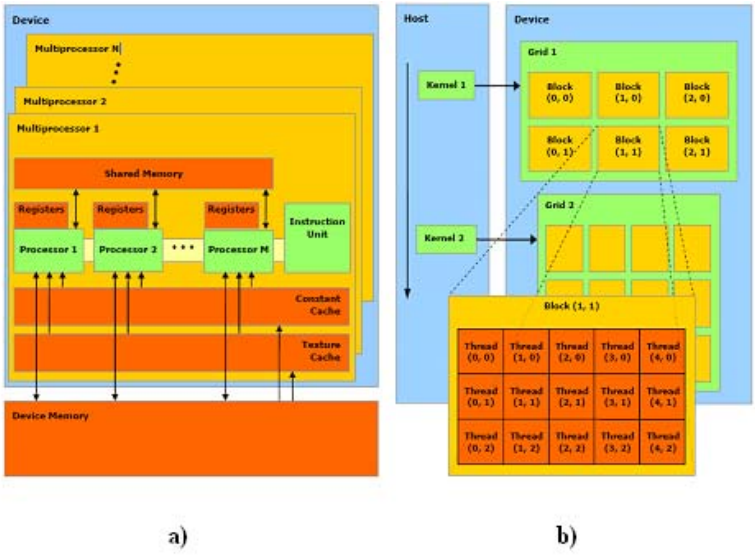

Figura 16 - a) esquema de multiprocessadores e de organização de memória do GPU, usando CUDA. b) modelo de programação e execução do CUDA.

Quando programado através do CUDA, o GPU é visto como uma máquina (*device*) capaz de executar um número muito elevado de *threads* (processos leves) em paralelo. O GPU pode, assim, ser visto como um co-processador ao CPU (*host*). Isto significa que uma porção de uma aplicação que seja executada muitas vezes de forma independente sobre dados diferentes no CPU, pode ser isolada para uma função, *kernel*, que o CPU (ou *host*) manda executar no GPU como muitos *threads* diferentes.

Cada *kernel* é organizado como uma grelha (*grid*) de blocos de *threads* (Figura 16 b)). Um bloco de *threads* é executado num único multiprocessador e consiste num conjunto de *threads* que podem cooperar entre si através da partilha de dados numa zona de memória partilhada de alta velocidade (um total de 16KB por multiprocessador [Cud07]).

O processo de simulação desenvolvido em [Birr07] que se pretende acelerar envolve, fundamentalmente, operações de cálculo vectorial e matricial. O método iterativo utilizado para a resolução do sistema linear de equações gerado pelo método implícito, é dominado fundamentalmente por operações de multiplicação entre matrizes esparsas e vectores densos, como será apresentado no próximo capítulo.

Tal como foi apresentado no 2º capítulo deste documento, os métodos de integração numérica têm um papel central em muitos problemas de simulação e, por serem métodos computacionalmente exigentes, são frequentemente candidatos a aceleração. O ponto fundamental será conseguir paralelizar estes algoritmos de uma forma adequada à arquitectura altamente paralela dos GPUs modernos.

Numa fase inicial, pretendeu-se estudar o potencial da aceleração da computação de operações de cálculo vectorial e matricial envolvidas no algoritmo que se pretende acelerar, com recurso ao modelo de programação CUDA, e como tal, devem ser salientadas algumas das considerações mais importantes a ter em conta quando se usa este modelo de programação. É também fundamental uma boa compreensão de alguns aspectos da arquitectura física dos GPUs modernos que ditam muita da performance deste modelo de programação.

3.1 Considerações sobre o modelo de programação:

O nvidia CUDA assenta num modelo de programação paralelo e escalável que permite misturar programação sequencial com programação paralela de uma forma heterogénea, encarando o GPU e o CPU como unidades de processamento distintas e com memórias diferentes, programáveis num ambiente de programação para computação paralela, através de extensões mínimas ao ambiente de programação C/C++.

Destas extensões, desde logo se destacam as declarações com o uso de prefixos que "estendem" o normal código C e que permitem especificar qual a zona de memória a que determinada estrutura, variável ou função pertence. Estes prefixos no código permitem instruir o compilador de CUDA, o nvcc, acerca do tipo de código a gerar:

- código CPU, que é posteriormente compilado pelo gcc/cl.
- código GPU que é posteriormente compilado pelo cudacc.

Uma outra extensão, é a introdução de um conjunto de palavras chave, usadas para aceder a variáveis especiais que são mapeadas pelo *driver* no *hardware*. Por exemplo, quando se acede à palavra chave *threadIdx*, na realidade, estamos a aceder a um registo no *hardware* que devolve ao *thread* o valor do seu identificador único.

O CUDA introduz ainda um conjunto de funcionalidades que se assemelham a chamadas de funções mas que são funcionalidades intrínsecas ao próprio *hardware*. Um bom exemplo é a chamada *__syncthreads()* que, na realidade, invoca sincronização por *hardware* do próprio GPU – *barrier syncronization* – ao nível de cada multiprocessador.

É também fornecida uma API de tempo de execução que fornece um conjunto de funções para gestão de memória e de execução de programas no GPU.

Uma outra consideração importante é a constatação de que o CUDA não assenta num modelo de programação apenas para GPU. Na realidade, grande parte do código CUDA executa no CPU, sempre que o programador considere que a computação desse código é mais adequada a executar em CPU – código não paralelizável. As porções de código CUDA que executam no GPU são, normalmente, pequenas porções de código altamente paralelizável, que são executadas por milhares de *threads* em simultâneo. Estas porções de código centram-se no paralelismo sobre os dados, procurando respeitar o modelo SIMD – *Single Instruction Multiple Data* [SIMD] – dos multiprocessadores (SMs – Stream Multiprocessors) do GPU. Respeitar o modelo SIMD obriga a que estruturas de grandes dimensões sejam particionadas em vários "pedaços" de pequena granularidade, que são então processados em paralelo segundo uma mesma instrução. No entanto, o modelo de programação em CUDA é, na realidade, um modelo SPMD – *Single Program Multiple Data* – e difere dos modelos SIMD no aspecto em que *threads* de um mesmo programa paralelo podem seguir caminhos divergentes – por exemplo, numa instrução do tipo *if...then...else*.

3.2 Considerações sobre a arquitectura

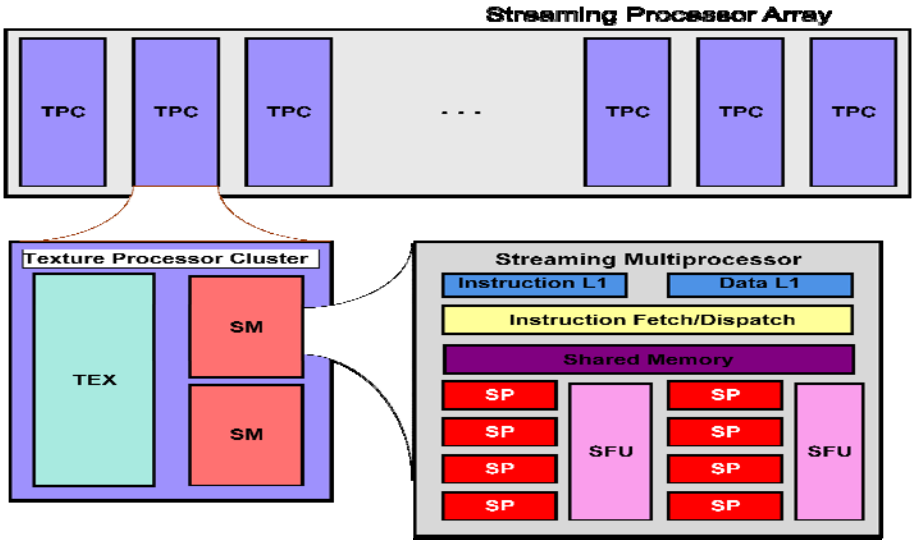

Figura 17 - Arquitectura física do NVIDIA G80 [Hwu07].

A figura (Figura 17) reflecte a realidade física do G80. Trata-se de um processador *stream* vectorial, composto por um conjunto de TPCs – *Texture Processor Clusters*. O número total de TPCs varia consoante o modelo do G80 em causa. Para as placas utilizadas no desenvolvimento desta dissertação, existem 6 TPCs no caso da GeForce 8800GTS e apenas 2 no caso da GeForce 8600M GS.

Independentemente do modelo do G80, cada TPC é composto por dois multiprocessadores stream (SM) e uma unidade de processamento de texturas (TEX), que é partilhada por ambos os SMs do TPC e que engloba a zona de cache de textura dos multiprocessadores CUDA (Figura 18).

Os SMs são, provavelmente, os componentes hardware mais importantes do GPU e são os responsáveis pelo processamento dos blocos de *threads* CUDA. A forma como estão internamente organizados dita muita da eficiência e das limitações do *hardware* e, consequentemente, do modelo de programação, como se verá mais à frente.

Cada SM respeita um modelo SIMD e é formado por um conjunto de componentes estre os quais se destacam uma cache de instruções (*Instruction L1*), uma cache de dados (*Data L1* – que se reflecte na zona de cache da memória constante do modelo CUDA), duas unidades de instruções (SFU – *Super Instruction Unit*) e uma zona de memória paralela (16 bancos de 1kB), partilhada por um conjunto de 8 processadores stream (SP) 32bit paralelos.

A cache de instruções é partilhada pelos 8 processadores, e é responsável por despachar uma mesma instrução para os 8 SPs em 1 clock. A cada ciclo de clocks, que na arquitectura actual equivale a 4 clocks, um SM executa então uma única instrução num grupo de 32 *threads*, a que se chama de *warp*. A este número fixo de *threads* que executam obrigatóriamente a mesma instrução SIMD, chama-se de tamanho do *warp*. A noção de *warp* é importante quando se pretende boa performance em tempo de execução. Um bom exemplo para compreender porquê, é um caso de divergências no caminho de execução de um *if...then...else*. Se houver divergência entre *threads* de *warps* diferentes, não há perdas de performance. No entanto, se dentro de um mesmo *warp* de *threads* houver divergência entre *threads*, então todo o *warp* de *threads* executa todo o *if...then...else*, havendo *threads* no *warp* que nada fazem no corpo do *then* e outros que nada fazem no corpo do *else*, fenómeno que resulta em perda de desempenho.

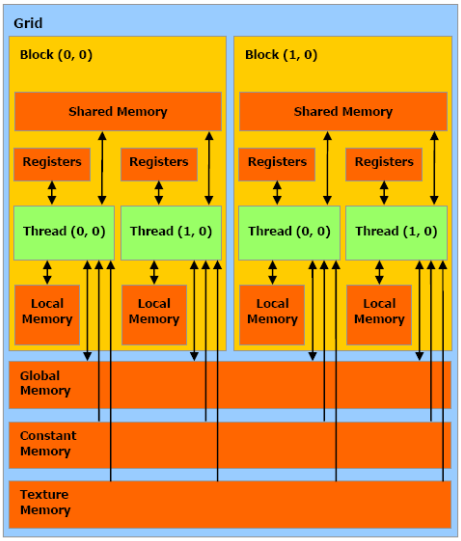

Figura 18 - modelo de memória CUDA.

Um outro aspecto fundamental é uma boa compreensão do funcionamento da memória gráfica (Figura 18). A zona de memória local é parte da memória global, mas local a um *thread* individual, ou seja, acessível apenas ao *thread* que a declara. As zonas de memória global, constante e de textura são de acesso muito mais lento que a zona de memória partilhada, que é memória paralela "onchip", dividida em 16 bancos de 1kB cada. Esta divisão da memória partilhada em bancos é essencial para se tirar bom partido do paralelismo dos processadores de cada SM. Assim, cada banco pode servir um endereço a um processador stream por cada *clock*, permitindo tantos acessos simultâneos por *clock* quanto o número total de bancos. No entanto, se existirem multiplos acessos simultâneos a um mesmo banco de memória, geram-se conflitos de bancos. Estes acessos em conflito são serializados, perdendo-se paralelismo. Assim, quanto maior o número de conflitos no acesso aos bancos de memória partilhada, maior a perda de performance, resultado dessas perdas de paralelismo no manuseamento da memória paralela.

Cada banco de 1k da memória partilhada tem uma largura de banda de 32bits por clock, ou seja, serve uma palavra de 32bits por clock. Palavras de 32bits consecutivas são atribuídas a bancos consecutivos. Tendo em conta que cada SM tem 16 bancos, que é o tamanho de meio *warp*, é garantido que apenas poderão haver conflitos dentro de um mesmo meio *warp* de threads [Hwu07]. Assim, a melhor forma de utilizar esta memória partilhada é, sempre que possível, garantir que todos os *threads* de cada meio *warp* acedem a bancos diferentes. Caso contrário, havendo acessos a um mesmo banco por multiplos *threads* de meio *warp*, esses acessos são serializados, sendo o custo determinado pelo número máximo de acessos simultâneos a um mesmo banco, nesse mesmo meio *warp*.

Tendo em conta esta memória partilhada e paralela de alta velocidade dos SMs, torna-se numa boa prática evitar os acessos a memória global, sempre que possível, particionando os dados em blocos mais pequenos que são carregados para a memória partilhada paralela de cada SM. Para tal, o particionamento dos dados deve ser feito para subconjuntos de dados que caibam no espaço de memória partilhada e cada subconjunto de dados deve ser gerido por um único bloco de *threads* da grid de execução, idealmente, conseguindo evitar conflitos de bancos.

Torna-se também relevante tirar partido da instrução de sincronização de *threads* do bloco para evitar erros nas computações em memória partilhada. Por exemplo, se tivermos um algoritmo em que cada *thread* de um bloco escreve um valor para memória partilhada e que posteriormente faz computações que envolvem também valores lidos por outros *threads* do bloco para a memória partilhada, se não houver uma instrução de sincronização ao nível do bloco entre as operações de leitura para memória partilhada e as operações de cálculo, podemos então estar a colocar o *thread* a tentar computar valores que ainda não foram escritos para memória partilhada. É então uma boa prática que, ao nível do bloco, todos os *threads* sincronizem após cada escrita para memória partilhada. No entanto, como será visível em alguns dos testes realizados, nem sempre se consegue tirar partido da memória de alta velocidade. De facto, existem dois tipos fundamentais de algoritmos altamente paralelos, aqueles que são limitados pela computação – *computation bound* – e os que são limitados pelos acessos a memória – *memory bound*. Neste ultimo caso, muitas vezes, a largura de banda da memória global dita a performance final do algoritmo.

O lançamento de funções para execução paralela no GPU exige que seja feita a configuração da execução no que toca ao número de blocos a executar na grid de execução e no que respeita ao número de *threads* por bloco. Tanto os *threads* como os blocos são associados a identificadores únicos (*threadIdx, blockIdx*) em tempo de execução, tornando possível utilizar essa informação para decidir qual a porção de dados sobre a qual cada *thread* irá trabalhar. Um bloco de *threads* pode ter 3 dimensões, xyz, sendo as dimensões máximas em cada eixo 512x512x64, respectivamente. No entanto, o número máximo de *threads* possíveis de executar num bloco de *threads* CUDA é 512.

O ciclo de vida de um *thread* CUDA no hardware começa com o lançamento de uma grid de execução. A grid é lançada para execução no SPA (*Stream Processor Array*) e os blocos de *threads* CUDA são distribuídos de forma sequencial para execução nos SMs sendo que, se houverem mais blocos de *threads* que SMs, os blocos passam a ser executados concorrentemente até um máximo de 8 blocos por SM, consoante os recursos alocados por cada bloco. Em cada SM, um bloco de *threads* CUDA é executado através do lançamento de *warps* de *threads* para execução. *Warps* e blocos que tenham terminado a sua execução libertam os seus recursos, possibilitando o lançamento de mais *warps* ou blocos no SM, caso necessário. Em cada SM, poderão estar até um máximo de 768 *threads* activos. Isto significa, por exemplo, que se cada bloco tiver 256 *threads*, estarão 3 blocos activos em cada SM, no máximo. Se o tamanho de bloco em número de *threads* for 128, então estarão 6 blocos activos em cada SM. No entanto, se forem definidos 64 *threads* por bloco, como cada SM gere um máximo de 8 blocos, apenas estarão 8*64=512 *threads* activos em cada SM.

Os *warps* são a unidade de despacho para execução dos SMs. Todos os *warps* cuja próxima instrução tenha os operandos prontos para processamento, ou seja, o operando na cache de instruções e os dados disponíveis, são considerados elegíveis para execução sendo depois seleccionados para execução de acordo com uma política de prioridades. Quando um *warp* é executado, todos os 32 *threads* do *warp* executam obrigatóriamente a mesma instrução, sendo necessários 4 *clocks* para o despacho da instrução a todo o *warp* de *threads*, como referido antes. Ora, cada acesso a memória global tem associada uma latência de 200 *clocks* o que significa que, se um acesso a memória for necessário por cada 4 instruções, será necessário um mínimo de 13 *warps* activos no SM para mascarar esta latência nos acessos a memória global.

O tamanho da grid, em número de blocos, pode ter apenas duas dimensões, xy. Aqui, as dimensões máximas são, na arquitectura actual, 65.535x65.535 respectivamente, sendo o máximo de blocos que podem ser atribuídos para execução num único GPU 65.535x65.535 = 4.294.836.225 blocos. A ideia prende-se com o facto de que a NVIDIA pretende atingir escalabilidade através da extensão, em futuras arquitecturas, não do número de *threads* de um bloco, mas sim deste número total de blocos possíveis de executar em cada dimensão de uma grid. Afinal, como vimos antes, o *hardware* dos SMs está limitado à gestão de um máximo de 768 *threads* distribuídos por um máximo de 8 blocos.

O *driver* de suporte a CUDA funciona de uma forma muito diferente dos *drivers* para gráficos. Trata-se de um *driver* optimizado para computação, cuja interface foi desenhada pela nvidia para computação “livre” de APIs gráficas. Mesmo assim, é garantida interoperabilidade com APIs gráficas como o OpenGL e o Directx, o que permite a utilização do *driver* para abordagens mistas, recorrendo ao CUDA para computação genérica e a APIs gráficas para a síntese de imagem. Isto representa uma vantagem no que toca, por exemplo, a simulações gráficas que envolvam muito cálculo, pois permite fazer os cálculos

mais pesados recorrendo ao CUDA e a posterior síntese de imagem recorrendo ao OpenGL ou ao Directx, tudo sem haver a necessidade de transferir dados entre a memória gráfica e a memória central.

3.3 NVIDIA CUBLAS

O CUBLAS[Cub07] é uma implementação do BLAS[BLAS] (*Basic Linear Algebra Subprograms*) sobre a plataforma CUDA, fornecendo suporte a uma série de operações sobre vectores e matrizes densos. O modelo de base pelo qual aplicações usam a biblioteca CUBLAS é através da criação de matrizes e vectores densos no espaço de memória do GPU (DRAM), enchendo-os com os dados inicializados do lado do CPU, chamar uma sequência de funções nucleares do CUBLAS e, finalmente, repor os resultados de volta para a memória central (RAM). Quanto maior for a sequência de operações possíveis de executar sobre os dados sem haver a necessidade de os retirar da DRAM, maiores serão os ganhos de performance da aplicação, por se conseguir minimizar a utilização do PCI-Express, o maior *bottle-neck* para computação genérica sobre GPU. Para tal, a biblioteca CUBLAS fornece ainda um conjunto de métodos auxiliares para a criação e destruição de objectos em memória gráfica, bem como para cópias de dados entre a RAM e a DRAM. Tendo em conta o estudo a realizar e estando esta biblioteca disponível, que implementa algumas das operações que se pretendem paralelizar, a comparação da performance conseguida nas *kernels* desenvolvidas com a dos métodos fornecidos no CUBLAS foi também objecto de estudo.

4. Análise do MPCG no CPU

No segundo capítulo foi apresentado o método de integração implícita de Euler, a propósito do trabalho desenvolvido em [Bar98] e [Birr07]. O objectivo deste capítulo é uma descrição mais pormenorizada sobre este método de integração numérica, bem como quais os requisitos da utilização de tal método de integração.

Se nos recordarmos, enquanto os métodos explícitos apenas se baseiam nas condições existentes no início do passo de integração, para avançar no tempo, o método implícito é expresso em termos das condições existentes no final do mesmo, tentando descobrir o próximo estado do sistema através da procura de um estado a partir do qual consiga recuar para o estado presente, retirando-lhe o passo de tempo.

Recordemos então a equação (2.8):

$$\begin{pmatrix} \Delta \boldsymbol{v} \\ \Delta \boldsymbol{p} \end{pmatrix} = \Delta t \begin{pmatrix} \boldsymbol{M^{-1}} \boldsymbol{f}(\boldsymbol{p_0} + \Delta \boldsymbol{p},\ \boldsymbol{v_0} + \Delta \boldsymbol{v}) \\ \boldsymbol{v_0} + \Delta \boldsymbol{v} \end{pmatrix} \qquad (4.1)$$

Relembrando, o vector $\boldsymbol{v_0}$ contém as velocidades de cada partícula no início do passo de simulação e $\boldsymbol{p_0}$ representa o vector das posições de cada partícula no mesmo instante. Os valores $\Delta \boldsymbol{v}$ e $\Delta \boldsymbol{p}$ representam as variações da velocidade e da posição do sistema para cada partícula, respectivamente. O passo de simulação é representado por Δt e $\boldsymbol{M^{-1}}$ é uma matriz diagonal que representa o inverso das massas de cada partícula, ou seja, para uma partícula i, $\boldsymbol{M_i}^{-1} = \frac{1}{m_i}$, sendo m_i a massa da partícula i. É então necessário que o sistema seja resolvido em ordem às incógnitas $\Delta \boldsymbol{v}$ e $\Delta \boldsymbol{p}$. No entanto, como referido no segundo capítulo, trata-se de um sistema não linear e a solução mais comum para o aproximar passa pela utilização da expansão em série de Taylor até às primeiras derivadas, resultando então:

$$\boldsymbol{f}(\boldsymbol{p_o} + \Delta \boldsymbol{p}, \boldsymbol{v_0} + \Delta \boldsymbol{v}) = \boldsymbol{f_0} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \Delta \boldsymbol{p} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} \Delta \boldsymbol{v} \qquad (4.2)$$

onde $\boldsymbol{f_0} = \boldsymbol{f}(\boldsymbol{p_0}, \boldsymbol{v_0})$ e as derivadas parciais são avaliadas para este estado inicial (p_0, v_0).

Usando esta aproximação, e usando a segunda linha da equação (4.1) para eliminar a variável $\Delta \boldsymbol{p}$ na primeira parcela do mesmo sistema, obtém-se a seguinte equação:

$$\Delta \boldsymbol{v} = \Delta t \boldsymbol{M}^{-1} \left(\boldsymbol{f_0} + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \Delta t (\boldsymbol{v_o} + \Delta \boldsymbol{v}) + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} \Delta \boldsymbol{v} \right) \qquad (4.3)$$

Se isolarmos $\Delta \boldsymbol{v}$, obtém-se

$$\left(\boldsymbol{I} - \Delta t \boldsymbol{M}^{-1} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} - \Delta t^2 \boldsymbol{M}^{-1} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \right) \Delta \boldsymbol{v} = \Delta t \boldsymbol{M}^{-1} \left(\boldsymbol{f_0} + \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \boldsymbol{v_0} \right) \qquad (4.4)$$

Esta equação apresenta no entanto um problema. A menos que todas as partículas no sistema possuam a mesma massa, a matriz do sistema não é simétrica. Para tornar a matriz do sistema simétrica e definida-positiva, basta multiplicar tudo pela matriz M, obtendo-se o sistema:

$$\left(\boldsymbol{M} - \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} - \Delta t^2 \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \right) \Delta \boldsymbol{v} = \Delta t \left(\boldsymbol{f_0} + \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} \boldsymbol{v_0} \right) \qquad (4.5)$$

Trata-se de um sistema linear, do tipo $Ax = b$, que pode agora ser resolvido por métodos iterativos como o método dos gradientes conjugados [She94], o qual permite obter uma boa solução em poucas iterações, sem modificar a matriz do sistema.

Resumindo, em termos operacionais, a utilização do método de integração implícita de Euler de primeira ordem, traduz-se na seguinte sequência de operações:

- Avaliar as forças e as derivadas parciais para o estado inicial;
- Formar o sistema da equação (4.5);
- Resolver o sistema em ordem a $\Delta \boldsymbol{v}$;
- Utilizar a solução obtida para actualizar as posições e velocidades das partículas.

É importante referir que, para um sistema com n partículas, os vectores $\boldsymbol{f_0}$ e $\boldsymbol{v_0}$ terão $3n$ valores, dadas as componentes x, y e z dos valores da força e da velocidade de uma partícula. Também, por consequência destas componentes, as derivadas parciais $\partial \boldsymbol{f} / \partial \boldsymbol{p}$ e $\partial \boldsymbol{f} / \partial \boldsymbol{v}$ são matrizes de dimensão $3n \times 3n$.

Ora, nos modelos cuja avaliação se faz com recurso a malhas de polígonos, como os descritos para os modelos de tecidos, há uma grande dependência dos vectores de força das partículas em relação a um número reduzido de partículas vizinhas. Como tal, o cálculo da força que actua sobre determinada partícula apenas está dependente das posições e velocidades de um grupo restrito de outras partículas, suas vizinhas. Deste modo, as matrizes das derivadas parciais resultam em matrizes de ocupação muito esparsa.

Assim, as estruturas de dados necessárias para a implementação do método implícito de integração devem oferecer suporte adequado a estas matrizes de ocupação esparsa suportando, simultaneamente, algoritmos eficientes para as operações matemáticas necessárias tais como o produto entre matrizes esparsas e vectores densos ou a adição de matrizes.

4.1 O método dos gradientes conjugados

O método dos gradientes conjugados é um método amplamente utilizado para a resolução iterativa de sistemas definidos positivos de equações lineares, como é o caso do sistema gerado pelo método implícito de Euler, sendo adequado à resolução de sistemas esparsos de grandes dimensões.

O método consiste na determinação de aproximações sucessivas à solução. A cada iteração, dois produtos internos são efectuados por forma a determinar escalares que são definidos por forma a permitir a satizfação de determinadas condições ortogonais. Em sistemas simétricos definidos positivos, estas condições implicam que a distância à verdadeira solução é minimizada.

O pseudo-código do método dos gradientes conjugados pré-condicionado é o apresentado na Listagem 1. Utiliza um pre-condicionador **M**, sendo que, para **M** = ***I***, o algoritmo resulta no método dos gradientes conjugados sem pre-condicionador.

Assim, dados como entrada uma matriz muito esparsa **A**, os vectores densos **x** e **b**, o pre-condicionador **M**, um número máximo de iterações i_{max} e um valor de tolerância de erro $\varepsilon < 1$, o sistema linear **Ax = b** pode ser resolvido seguindo o algoritmo descrito na listagem seguinte:

$$
\begin{aligned}
&i \leftarrow 0;\ \mathbf{r} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x};\ \mathbf{d} \leftarrow \mathbf{M}^{-1}\mathbf{r};\\
&\delta_{new} \leftarrow \mathbf{r}^T\mathbf{d};\ \delta_0 \leftarrow \delta_{new};\\
&\textbf{while } i < i_{max} \textbf{ and } \delta_{new} > \varepsilon^2\delta_0 \textbf{ do}\\
&\qquad \mathbf{q} \leftarrow \mathbf{A}\mathbf{d};\ \alpha \leftarrow \frac{\delta_{new}}{\mathbf{d}^T\mathbf{q}};\\
&\qquad \mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{d};\ \mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q};\\
&\qquad \mathbf{s} \leftarrow \mathbf{M}^{-1}\mathbf{r};\ \delta_{old} \leftarrow \delta_{new};\\
&\qquad \delta_{new} \leftarrow \mathbf{r}^T\mathbf{s};\ \beta \leftarrow \frac{\delta_{new}}{\delta_{old}};\\
&\qquad \mathbf{d} \leftarrow \mathbf{r} + \beta\mathbf{d};\ i \leftarrow i + 1;\\
&\textbf{end}
\end{aligned}
$$

Listagem 1 - pseudo-código para o algoritmo dos gradientes conjugados pre-condicionado (PCG).

A nível operacional, o método dos gradientes conjugados envolve, por cada iteração, dois produtos entre uma matriz esparsa e um vector denso, três operações sobre vectores (soma, subtracção ou multiplicação por um escalar) e dois produtos internos.

4.2 O método dos gradientes conjugados pre-condicionado modificado

Uma das grandes contribuições do trabalho de Baraff e Witkin em [Bar98], foi a proposta de um método de resolução para o sistema de equações do método implícito de Euler, derivado do método dos gradientes conjugados. Este método permite aplicar um conjunto de restrições ao movimento das partículas, apresentando a vantagem de não prejudicar o desempenho do algoritmo e garantindo integralmente o respeito das restrições impostas.

A solução adoptada foi a de modificar o método dos gradientes conjugados pré-condicionado por forma a que este aplique as restrições desejadas a cada iteração, através de operações de filtragem sobre a solução, descartando assim as mudanças proibidas. Desta forma, o método permite que, independentemente do número de iterações efectuadas, as restrições sejam sempre impostas na totalidade.

O algoritmo da listagem seguinte (Listagem 2) ilustra o método dos gradientes conjugados pré-condicionado modificado (MPCG) proposto por Baraff e Witkin em [Bar98].

Argumentos:
$\mathbf{A}$ - matriz do sistema linear
$\mathbf{P}$ - Pré-condicionador do sistema
$\mathbf{S}$ - Matriz diagonal definida por $diag\{\mathbf{S}_1, ..., \mathbf{S}_n\}$
$\mathbf{b}$ - termo independente do sistema linear
$\mathbf{z}$ - alterações de velocidade desejadas para as partículas restringidas
Resultado:
$\Delta\mathbf{v}$ - solução do sistema

1: $\Delta\mathbf{v} = \mathbf{z}$
2: $\delta_0 = (\mathbf{Sb})^T\mathbf{P}(\mathbf{Sb})$
3: $\mathbf{r} = \mathbf{S}(\mathbf{b} - \mathbf{A}\Delta\mathbf{v})$
4: $\mathbf{c} = \mathbf{SP}^{-1}\mathbf{r}$
5: $\delta = \mathbf{r}^T\mathbf{c}$
6: **while** $\delta > \varepsilon^2\delta_0$ **do**
7: $\quad\mathbf{q} = \mathbf{SAc}$
8: $\quad\alpha = \delta/(\mathbf{c}^T\mathbf{q})$
9: $\quad\Delta\mathbf{v} = \Delta\mathbf{v} + \alpha\mathbf{q}$
10: $\quad\mathbf{r} = \mathbf{r} - \alpha\mathbf{q}$
11: $\quad\mathbf{s} = \mathbf{P}^{-1}\mathbf{r}$
12: $\quad\delta_{old} = \delta$
13: $\quad\delta = \mathbf{r}^T\mathbf{s}$
14: $\quad\mathbf{c} = \mathbf{S}(\mathbf{s} + \frac{\delta}{\delta_{old}}\mathbf{c})$
15: **end while**

Listagem 2 - pseudo-código para o algoritmo dos gradientes conjugados pre-condicionado modificado (MPCG).

No trabalho de Baraff e Witkin [Bar98], é descrita também a necessidade de um procedimento auxiliar, cuja função é a filtragem de vectores. Esta operação de filtragem é equivalente a um produto matriz-vector, recorrendo a uma matriz de filtragem, **S**, que é aplicada ao vector a filtrar. Este algoritmo difere do PCG, em primeiro lugar, no aspecto em que a estimativa inicial para a solução consiste em usar o vector com as modificações de velocidade propostas para as partículas restringidas. Um outro aspecto reflecte-se na presença da matriz de filtragem **S**, com a qual se efectuam modificações de modo a manter as restrições. Também o critério de paragem difere, na medida em que tem em consideração que algumas das componentes do vector **b** são ignoradas pelo algoritmo, devendo, portanto, ser também ignoradas para determinar o fim das iterações. Finalmente, dado que o vector **r** serve para medir o erro da solução, este não deverá ser afectado pelas restrições.

O MPCG utilizado em [Birr07] e que se pretende acelerar com recurso ao GPU, apresenta ainda uma modificação ao método proposto por Baraff e Witkin, introduzida por Ascher e Boxerman [Asc03]. Trata-se de uma versão corrigida do método modificado, cuja principal característica rezide no facto de permitir utilizar a solução anterior na fase de inicialização do algoritmo, como forma de o acelerar, reduzindo o número de iterações necessárias.

Com esta correcção, em vez de utilizar como solução inicial o valor de **z** (ver linha 1 da listagem 4.2-1), o MPCG incorpora o resultado da iteração anterior para acelerar a convergência, explorando a coerência temporal do sistema [Birr07]. A inicialização de $\Delta \boldsymbol{v}$, na linha 1 da listagem 4.2-1, passa a ser dada por $\Delta \boldsymbol{v} = \boldsymbol{S}\Delta \boldsymbol{v}_0 + (\boldsymbol{I} - \boldsymbol{S})\boldsymbol{z}$, sendo $\Delta \boldsymbol{v}_0$ a solução da iteração anterior. Além desta alteração, é também utilizado um critério de paragem diferente.

4.3 Paralelismo

Torna-se então relevante discernir quais as operações envolvidas na resolução do método dos gradientes conjugados modificado utilizado em [Birr07] por forma a identificar quais as possibilidades de paralelizar tal algoritmo para processamento paralelo no GPU.

O GPU moderno é um processador altamente paralelo e com enorme largura de banda, cujos processadores respeitam um modelo SIMD – Single Instruction Multiple Data. Quando utilizado para computação genérica, o GPU actua como um co-processador ao CPU (Figura 19).

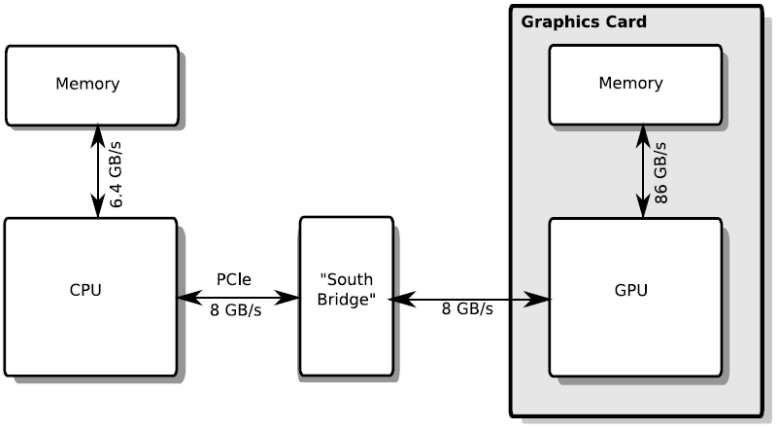

Figura 19 - esquema típico de um sistema que use o GPU como um co-processador.

O GPU comunica com o CPU através do bus PCI-Express. O PCIe 16x é capaz de alcançar 4GB/s na transferência de dados em cada sentido logo, o pico máximo de largura de banda na comunicação com o CPU é de 8 GB/s, enquanto o pico máximo da largura de banda da memória gráfica é, pelo menos, uma ordem de magnitude, ou várias, acima da largura de banda do PCI-Express. Assim, o PCI-Express pode representar um sério *bottleneck* se a intensidade aritmética dos cálculos efectuados no GPU não for suficientemente pesada.

Ora, tal como identificado em [Bar94], é facil verificar que o MPCG é um método fundamentalmente dominado pelo seguinte conjunto de operações:

- Operações sobre vectores densos como soma, subtracção ou multiplicação por escalares;
- Produtos internos;
- Produtos entre matrizes esparsas e vectores.

As operações sobre vectores (soma, subtracção ou multiplicação por escalares) são operações trivialmente paralelizáveis, por exemplo, colocando cada processador a fazer a operação sobre o seu próprio segmento. Afinal, a computação de cada índice do vector resultado pode ser computada de forma independente de todas as outras.

A computação de produtos internos é facilmente paralelizável, colocando cada processador a calcular o produto interno dos segmentos que lhe forem atribuídos, produtos internos locais. Os resultados destes produtos internos locais devem depois ser reduzidos a um único resultado global.

A operação de multiplicação entre matrizes esparsas e vectores é também uma operação paralelizável, bastando dividir a matriz em linhas que correspondam a segmentos do vector resultado e colocando cada processador a computar o produto da linha da matriz pelo vector.

Como todas as operações que dominam o método dos gradientes conjugados são operações fácilmente paralelizáveis, estamos perante um algoritmo com um padrão embaraçosamente paralelo [Mas99]. Torna-se assim possível a perspectiva de executar a totalidade do algoritmo no espaço de memória do GPU, evitando ao máximo a utilização do bus PCIe, o maior *bottleneck* na utilização do GPU como um co-processador ao CPU.

5. MPCG no GPU

Como vimos no 2º capítulo deste documento, as abordagens anteriores para aceleração da animação de superfícies deformáveis, tais como os tecidos, com recurso à programação do GPU, recorrem fundamentalmente a APIs como o GLSL ou o HLSL, linguagens de *shading* do OpenGL e do DirectX, respectivamente, para implementar operações paralelas. Conforme foi apresentado nesse mesmo capítulo, esse modelo de programação, centrado nos gráficos, limita a flexibilidade de programação, a performance e as possibilidades dos GPUs modernos em termos de GPGPU.

Foi também discutido, no capítulo anterior, o algoritmo que se pretende implementar. Este requer um conjunto de operações BLAS sobre vectores densos, estruturas adequadas para a representação de matrizes esparsas e suporte a operações de multiplicação entre matrizes esparsas e vectores densos. Assim, neste capítulo, apresenta-se um estudo sobre diferentes abordagens para a implementação em CUDA, apresentado em pormenor no 3º capítulo, de cada operação BLAS necessária ao MPCG, sendo apresentados os resultados obtidos para cada operação, bem como o speedup obtido para a melhor abordagem, face ao tempo para a mesma operação em CPU.

Como referido anteriormente, a NVIDIA disponibiliza uma biblioteca para operações BLAS sobre vectores e matrizes densos – o CUBLAS. Um dos objectivos, no estudo de diferentes abordagens para a implementação das operações BLAS necessárias ao MPCG, foi também o de comparar o desempenho entre as operações do CUBLAS com as melhores implementações conseguidas.

Para a realização do estudo, o CPU e os GPUs utilizados estão apresentados na tabela seguinte, onde são também reveladas algumas das características *hardware*:

	CPU	GPU 1	GPU 2
Processador	Intel Core 2 Duo T8300	GeForce 8600 M GS	GeForce 8800 GTS Fatal1ty
Código	Penryn	G86 M	G80
Frequência de clock	2.40 GHz	500 MHz	650 MHz
Memória	4 GB DDR2	512 MB DDR2	320 MB GDDR3
Largura de banda máxima	6.0 GB/s	12.8 GB/s	80.0 GB/s
Velocidade de memória	2 x 333 MHz	2 x 400MHz	2 x 1.0GHz
Número de núcleos	2	32	96
Largura do Bus	64 bit	128 bit	320 bit

Tabela 1 - características do CPU e GPUs utilizados no desenvolvimento do trabalho.

Numa fase inicial, realizou-se um estudo com o objectivo primário de verificar quais os ganhos reais de paralelizar o conjunto de operações vectoriais e matriciais utilizados no MPCG, recorrendo ao CUDA. O estudo consistiu na comparação dos tempos médios de execução entre CPU e GPUs para cada operação, com volumes de dados diferentes. Supostamente, a biblioteca CUBLAS fornece a implementação GPU mais eficiente de BLAS (*Basic Linear Algebra Sub-programs*) em CUDA para estruturas densas e, como tal, um segundo objectivo foi verificar os tempos de execução, largura de banda aproveitada e performance dos métodos CUBLAS e compará-los, também, com os tempos obtidos nas *kernels* desenvolvidas que se revelaram mais eficientes.

O facto de haver dois GPUs com características diferentes disponíveis para teste, permitiu ainda verificar a escalabilidade de cada algoritmo entre arquitecturas (G86M vs G80). Os resultados obtidos permitem, assim, uma escolha fundamentada sobre quais os métodos a utilizar, ou a desenvolver, por forma a atingir os objectivos propostos neste documento. Todos os tempos apresentados são a média de 100 execuções consecutivas de cada algoritmo.

5.1 Soma e subtracção de vectores no GPU

Em código sequencial (CPU), este tipo de operações são realizadas através dum simples ciclo *for* que itera as posições dos vectores e, para cada posição, soma ou subtrai os valores dessa posição nos vectores de entrada, escrevendo o resultado da operação na mesma posição no vector de saída, como se vê na listagem seguinte:

```
for( unsigned int i = 0; i < len; ++i){
        c_out[i] = a_in[i] + b_in[i];
}
```

Listagem 3 - código sequencial CPU para soma de vectores.

Trata-se dum algoritmo simples e de complexidade temporal *O(n)*, onde *n* é o número de elementos dos vectores. Este algoritmo é facilmente paralelizável usando CUDA. Se cada *thread* do algoritmo paralelo executar o trabalho equivalente a uma das iterações do ciclo *for*, torna-se possível fazer o mesmo trabalho com um algoritmo de complexidade temporal *O(n/p)*, sendo o *p* o número de processadores stream do GPU.

A grid pode ser configurada de forma a adequar-se ao tipo de problema e, tratando-se de um problema de solução linear, define-se uma grid também linear. Sabendo que uma boa ocupação dos SMs é adequada a um bom aproveitamento do GPU, e uma vez que se pretende realizar testes sobre dados de grandes dimensões, deve ser criada uma grid de execução que lance um total de threads igual ou superior ao número de elementos do vector. Isto é adequado para execução no GPU pois implica o lançamento de milhares de threads. Para se poderem fazer testes com vectores de grandes dimensões, optou-se por definir os blocos como blocos vectoriais de 512 *threads*. Para saber qual o trabalho a realizar, cada *thread* recorre às palavras-chave *threadIdx*, *blockIdx e blockDim*. No caso específico destes algoritmos, o índice *i* a computar é obtido com base nestas palavras chave. Se o índice obtido estiver dentro dos limites do tamanho dos vectores, o *thread* efectua a computação, caso contrário nada faz, como se mostra na listagem seguinte:

```
__global__ void vecSumKernel_1( float* a_in, float* b_in, float* c_out, int size) {

   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i<size){
           c_out[i] = a_in[i] + b_in[i];
   }
}
```

Listagem 4 – código da primeira kernel CUDA desenvolvida para soma de vectores.

Apenas é feita uma única computação, uma soma ou subtracção, para cada três acessos a memória – dois acessos para as posições de leitura nos vectores a_in e b_in e um acesso para a posição de escrita em c_out. Trata-se então de um algoritmo limitado pela largura de banda da memória gráfica, pois por cada operação realizada (flop) são necessários 2 acessos de leitura e um de escrita para a DRAM. Nestas situações, a abordagem ideal passa por aproveitar a capacidade do GPU de fazer computação para memória (*compute to memory*) tentando maximizar o aproveitamento da largura de banda disponível.

Soma de vectores	8600M GS (ms)	GB/s	GFLOPS	8800 GTS (ms)	GB/s	GFLOPS
kernel 1 – 100000 elementos	0,231	4,95	0,43	0,049936	22,92	2,00
kernel 1 – 1000000 elementos	1,95912	5,84	0,51	0,259214	44,15	3,86
kernel 1 – 10000000 elementos	19,267279	5,94	0,52	2,012189	56,87	4,97

Tabela 2 – tempos de execução médios da primeira kernel CUDA, aproveitamento da largura de banda (GB/s) e número de operações de vírgurla flutuante por segundo (GFLOPS).

Como é visível na Tabela 2, em nenhum dos testes se consegue aproveitar o máximo da largura de banda disponível nos GPUs. Além disso, tendo em conta que estamos a utilizar blocos de 512 *threads*, apenas um único bloco é atribuído a cada SM o que significa que apenas se está a tirar partido de um único nível de paralelismo, o número total de processadores.

Assim, dada a disponibilidade para maior aproveitamento da largura de banda da memória gráfica, podemos tentar optimizar o algoritmo obrigando cada *thread* a fazer 2 computações em vez de uma. No entanto, ao fazer esta optimização, é importante ter em conta a coalescência nos acessos a memória global. O princípio de coalescência nos acessos a memória global diz-nos que, se todos os *threads* de um *warp* acederem a posições contíguas de memória, é garantido o paralelismo entre todos os *threads* nesse acesso, caso contrário, os acessos a memória serão serializados, obrigando à repetição de *warps* inteiros por cada acesso não contíguo. Assim, a segunda posição lida de memória global num *thread* de um bloco deve no mínimo ser afastada a dimensão do *warp* em relação à primeira posição lida por esse mesmo *thread*, por forma a garantir que os restantes *threads* do *warp* conseguem ler todas as posições contíguas, garantindo assim a coalescência. Para garantir a coalescência no acessos a memória optou-se, então, por espaçar os endereços lidos em cada *thread*, pela dimensão do bloco, como apresentado na listagem seguinte:

```
__global__ void vecSumKernel_2( float* a_in, float* b_in, float* c_out, int size) {

   int i = threadIdx.x + (blockDim.x*2) * blockIdx.x;
        int i2 = i + blockDim.x;
   if (i<size){
           c_out[i] = a_in[i] + b_in[i];
   }
   if(i2<size){
           c_out[i2] = a_in[i2] + b_in[i2];
   }
}
```

Listagem 5 – código da segunda kernel CUDA desenvolvida para soma de vectores.

Esta abordagem permite ainda reduzir para metade o número blocos a executar na grid, mantendo o tamanho do bloco. Se compararmos os resultados obtidos entre as duas *kernels* verifica-se um ligeiro ganho de performance (Tabela 3), relacionado com o melhor aproveitamento da largura de banda da memória gráfica e com o menor número de blocos a executar.

Soma de vectores	8600M GS (ms)	GB/s	GFLOPS	8800 GTS (ms)	GB/s	GFLOPS
kernel 2 – 100000 elementos	0,2115	5,41	0,47	0,048319	23,68	2,07
kernel 2 – 1000000 elementos	1,66174	6,89	0,60	0,236364	48,42	4,23
kernel 2 – 10000000 elementos	16,43018	6,97	0,61	1,924199	59,47	5,20

Tabela 3 – tempos de execução médios da segunda kernel CUDA, aproveitamento da largura de banda (GB/s) e número de operações de vírgurla flutuante por segundo (GFLOPS).

Esta optimização deveria apresentar ganhos de performance mais evidentes mas, uma vez que cada bloco tem 512 *threads*, ainda não estamos a tirar partido do segundo nível de paralelismo disponível em CUDA, pois apenas 1 único bloco está activo em cada SM em simultâneo.

Tendo em conta que ainda não estamos a utilizar o máximo da largura de banda disponível, optou-se por obrigar que cada *thread* some então 4 posições, reduzindo para metade o número de *threads* por bloco em relação à implementação anterior, como se vê na listagem seguinte:

```
__global__ void vecSumKernel_3( float* a_in, float* b_in, float* c_out, int size) {

   int i = threadIdx.x + (blockDim.x*4) * blockIdx.x;
   int i2 = i + blockDim.x;
   int i3 = i2 + blockDim.x;
   int i4 = i3 + blockDim.x;
   if (i<size){
           c_out[i] = a_in[i] + b_in[i];
   }
   if(i2<size){
           c_out[i2] = a_in[i2] + b_in[i2];
   }
        if (i3<size){
           c_out[i3] = a_in[i3] + b_in[i3];
   }
   if(i4<size){
           c_out[i4] = a_in[i4] + b_in[i4];
   }
}
```

Listagem 6 – código da terceira kernel CUDA desenvolvida para soma de vectores.

Como podemos observar na tabela abaixo (Tabela 4), os ganhos conseguidos com esta abordagem são semelhantes aos da primeira abordagem, mas ligeiramente superiores, apesar de seguir o mesmo princípio. Este ganho de performance não se deve apenas ao maior número de somas por *thread*. Na realidade, ao utilizarmos 256 *threads* por bloco, uma vez que cada SM pode gerir um máximo de 768 *threads*, cada SM passa a ter 3 blocos activos em simultâneo em vez de apenas um, como nos casos anteriores. Assim, passamos a ter 2 níveis de paralelismo, o primeiro ao nível do número de blocos por SM e o segundo, já presente nos testes anteriores, ao nível do número de processadores disponíveis. O aproveitamento destes dois níveis de paralelismo, permitem disfarçar melhor a latência nos acessos a memória.

Soma de vectores	8600M GS (ms)	GB/s	GFLOPS	8800 GTS (ms)	GB/s	GFLOPS
kernel 3 – 100000 elementos	0,18864	6,07	0,53	0,046446	24,64	2,15
kernel 3 – 1000000 elementos	1,55702	7,35	0,64	0,216008	52,98	4,63
kernel 3 – 10000000 elementos	15,101679	7,58	0,66	1,775171	64,47	5,63

Tabela 4 – tempos de execução médios da terceira kernel CUDA, aproveitamento da largura de banda (GB/s) e número de operações de vírgurla flutuante por segundo (GFLOPS).

Como referido antes, um outro objectivo do estudo passou por testar e comparar as performances conseguidas, com as obtidas pelos métodos implementados na biblioteca CUBLAS da NVIDIA. Esta biblioteca fornece um método, o cublasSaxpy(), através do qual é possível fazer a soma ou a subtracção de vectores densos. Os tempos obtidos ao recorrer a este método são os apresentados na tabela seguinte:

Soma de vectores	8600M GS (ms)	GB/s	GFLOPS	8800 GTS (ms)	GB/s	GFLOPS
cublasSaxpy() – 100000 elementos	0,19818	5,77	0,50	0,044874	25,50	2,23
cublasSaxpy() – 1000000 elementos	1,574	7,27	0,64	0,275175	41,59	3,63
cublasSaxpy() – 10000000 elementos	15,10984	7,57	0,66	2,603387	43,96	3,84

Tabela 5 – tempos de execução médios do método CUBLAS cublasSaxpy(), aproveitamento da largura de banda (GB/s) e número de operações de vírgurla flutuante por segundo (GFLOPS).

Se analisarmos os resultados obtidos, na Nvidia GeForce 8600M GS notamos que o melhor resultado conseguido foi conseguir aproximar os resultados obtidos com o método fornecido pelo CUBLAS, quer no que toca a aproveitamento da largura de banda da memória gráfica, quer no que toca à performance, atingindo o máximo de 0,66 GFLOPS para o teste mais pesado. No entanto, se observarmos os resultados conseguidos na Nvidia GeForce 8800 GTS, conseguimos tempos e performances muito semelhantes para volumes de dados mais pequenos, como no caso do 1º teste, mas, para maiores volumes de dados, a *kernel* 3 ultrapassa claramente a performance do cublasSaxpy(), atingindo um pico de 5,63 GFLOPS contra o pico de 3,84 GFLOPS no cublas, quase menos 2 GFLOPS. Certamente, este resultado é reflexo do maior aproveitamento da largura de banda na *kernel* 3, um pico de 64,47GB/s contra os 43,96GB/s conseguidos pelo cublas. Esta observação dá a ideia que o algoritmo desenvolvido para a *kernel* 3 escala melhor com o aumento da largura de banda disponível e do número de processadores do que o algoritmo implementado no CUBLAS.

Em relação aos tempos de CPU para os mesmos testes, segundo o algoritmo sequencial apresentado na Listagem 3, os resultados são os apresentados na Tabela 6. Os speedups apresentados dizem respeito à *kernel* 3, quando comparada com o CPU, na GeForce 8800 GTS.

Soma de vectores	Core2Duo 2.4GHz (ms)	GeForce 8800 GTS (ms)	speedup
teste 1 – 100000 elementos	0,1347	0,046446	2,90 x
teste 2 – 1000000 elementos	3,727	0,216008	17,25 x
teste 3 – 10000000 elementos	35,810902	1,775171	20,17 x

Tabela 6 – speedups conseguidos com a terceira kernel implementada para somas, face aos tempos de CPU para os mesmos testes.

Os resultados obtidos para as operações de subtracção foram muito semelhantes aos obtidos para soma de vectores, como se vê na tabela abaixo. No fundo, foi seguido o mesmo esquema de optimizações tomado para as somas. As soluções desenvolvidas para subtracção de vectores são assim, logicamente, em tudo idênticas às soluções desenvolvidas para somas, pelo que não se torna relevante uma descrição pormenorizada, como foi feito para as somas.

Subtracção de vectores	Core2Duo 2.4GHz (ms)	GeForce 8800 GTS (ms)	speedup
teste 1 – 100000 elementos	0,1078	0,053522	2,01 x
teste 2 – 1000000 elementos	3,8039	0,211781	17,96 x
teste 3 – 10000000 elementos	35,510597	1,769955	20,06 x

Tabela 7 – speedups conseguidos com a terceira kernel implementada para subtracções, face aos tempos de CPU para os mesmos testes.

5.2 Produtos internos

O produto interno entre vectores é uma operação que consiste no somatório de todas as multiplicações entre os elementos de mesmo índice dos vectores, ou seja, $c = \sum_0^n a_i \times b_i$

Esta operação pode ser conseguida de forma relativamente simples em código sequencial, com recurso a um simples ciclo *for* que itera todos os índices dos vectores de entrada, acumulando para retorno o resultado das multiplicações entre os valores de cada índice, como representado na listagem seguinte:

```
float computeVecDot(const float* a_in, const float* b_in, const unsigned int len) {
    float dot = 0.0f;
    for( unsigned int i = 0; i < len; ++i) {
        dot += a_in[i] * b_in[i];
    }
    return dot;
}
```

Listagem 7 – código sequencial CPU para produtos internos entre vectores.

Trata-se de um algoritmo de complexidade temporal *O(n)*, onde *n* é o número de elementos nos vectores de entrada, havendo um total de 2*n* operações de vírgula flutuante, uma multiplicação e uma soma por cada um dos *n* índices.

Na primeira abordagem para a paralelização desta operação, dividiu-se a operação em dois passos:

- A multiplicação directa entre os valores dos vectores.
- A redução do vector resultado através da soma de todos os seus elementos.

A operação de multiplicação directa entre os valores dos vectores é facilmente paralelizável, bastando seguir o mesmo princípio adoptado para a soma de vectores e substituíndo o operador de soma pelo operador de multiplicação, obtendo um vector com o resultado de todas as multiplicações. No entanto, pretende-se ainda obter o valor da soma de todos os elementos deste vector resultado. Assim, uma vez que se pretende somar os valores contidos em todos os índices do vector das multiplicações para um único endereço, faz aqui sentido tirar partido da memória partilhada de alta velocidade, para fazer a redução de sub-vectores com os valores das multiplicações, por forma a evitar interacção com a memória global,

procurando assim reduzir a latência. Sabemos que há um número limitado de *threads* por cada bloco CUDA e que a memória paralela apenas é partilhada pelos *threads* de um único bloco. Assim, em cada bloco, todos os *threads* fazem multiplicações para endereços de memória partilhada. Depois, ao nível desse bloco, é reduzido o sub-vector em memória partilhada, com os resultados das multiplicações, a um único valor.

Uma vez que não existem funções de sincronização entre blocos, cada bloco escreve um resultado parcial, um único valor, num endereço diferente de memória global. O CPU é depois utilizado para somar todos os resultados parciais, tantos quantos os blocos lançados na *grid* de execução.

É importante perceber como é feita a soma dos valores em memória partilhada, que acontece no ciclo *for* da listagem seguinte (Listagem 8). Este ciclo obriga a que a operação de soma seja feita com endereçamento sequencial em memória partilhada que, como foi explicado antes, é um tipo de endereçamento que é livre de conflitos nos acessos aos bancos de memória partilhada. A cada passo do ciclo, o número de *threads* activos no bloco vai reduzindo.

```
__global__ void vecDotKernel_1( float* a_in, float* b_in, float* c_out, int size) {
   //declaração da memória partilhada
    SharedMemory<float> smem;
    float *sdata = smem.getPointer();
    // obter o índice a multiplicar para memória partilhada
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   // se o índice estiver dentro dos limites fazer a multiplicação
    if(i<size){
           sdata[tid] = a_in[i] * b_in[i];
    } else{
           sdata[tid] = 0.0f;
    }
   // sincronizar para garantir que todos os threads realizaram a operação para memória
partilhada
    __syncthreads();

    // fazer a reducao parcial em memoria partilhada - soma das multiplicações do bloco
    for(unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
          // Sincronizar para garantir que o passo de redução seguinte pode ocorrer
        __syncthreads();
    }

    // escrever o resultado do bloco para memória partilhada
    if (tid == 0) c_out[blockIdx.x] = sdata[0];
}
```

Listagem 8 – código da primeira kernel CUDA desenvolvida para produtos internos entre vectores.

Produto interno	8600M GS (ms)	GB/s	GFLOPS	8800 GTS (ms)	GB/s	GFLOPS
kernel 1 – 100000 elementos	1,18444	0,64	0,17	0,534691	1,43	0,37
kernel 1 – 1000000 elementos	10,016599	0,76	0,20	3,648793	2,09	0,55
kernel 1 – 10000000 elementos	96,711189	0,79	0,21	33,097023	2,31	0,60

Tabela 8 – tempos de execução médios da primeira kernel CUDA, aproveitamento da largura de banda (GB/s) e número de operações de vírgurla flutuante por segundo (GFLOPS).

Na tabela acima (Tabela 8), são apresentados os tempos médios de execução da 1ª *kernel* para produtos internos em GPU. É fácil verificar que a solução está longe de ser uma opção viável pela nítida morosidade de execução. Os tempos apresentados implicam não apenas execução de código GPU, mas também a transferência dos resultados parciais através do PCI-Express e ainda um ciclo CPU para efectuar a redução final desses resultados. Torna-se nítido o *bottleneck* que o PCI-Express representa, pelo que esta implementação não se revela uma boa opção para a paralelização dos produtos internos. Há a necessidade de evitar o uso do PCI-Express, ou seja, de efectuar todo o calculo no GPU.

Ora, seguindo o raciocínio tomado para a soma e subtracção de vectores, esta primeira *kernel* pode ser optimizada por forma a conseguir um melhor aproveitamento da largura de banda gráfica. Essa abordagem permite reduzir assim o total de blocos para metade, bem como o tamanho dos blocos para 256 *threads*, permitindo assim explorar os 2 níveis de paralelismo. Além disto, é sabido [Cud07] que a pesença de ciclos no código paralelo GPU tem implicações negativas na performance dos algorimtos, pelo que se pode também «desfazer» o ciclo *for* presente na *kernel* aplicando a técnica de *loop unrolling*.

Este ciclo pode ser «desfeito» de uma forma genérica pelo facto de sabermos que o tamanho máximo de um bloco CUDA é de 512 *threads*, e por sabermos que todos os tamanhos de bloco com que trabalhamos são múltiplos de 2. Assim, resulta a listagem seguinte:

```
__global__ void vecDotKernel_2( float* a_in, float* b_in, float* c_out, int size) {
    SharedMemory<float> smem;
    float *sdata = smem.getPointer();

// em vez de uma única multiplicação para memória partilhada,
//cada thread do bloco faz até 4 somas de 4 multiplicações dos vectores de entrada
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*4) + threadIdx.x;
    unsigned int i2 = i+blockDim.x;
    unsigned int i3 = i2+blockDim.x;
    unsigned int i4 = i3+blockDim.x;
    if(i4<size){
          sdata[tid] += (a_in[i] * b_in[i]) + (a_in[i2] * b_in[i2]) + (a_in[i3] * b_in[i3])
                      + (a_in[i4] * b_in[i4]);
    } else if(i3<size && i4>=size){
          sdata[tid] += (a_in[i] * b_in[i]) + (a_in[i2] * b_in[i2]) + (a_in[i3] * b_in[i3]);
    } else if(i2<size && i3>=size){
          sdata[tid] += (a_in[i] * b_in[i]) + (a_in[i2] * b_in[i2]);
    } else if(i<size && i2>=size){
          sdata[tid] += (a_in[i] * b_in[i]);
    } else{
          sdata[tid] += 0.0f;
    }
    __syncthreads();
```

```
// cada bloco tem um máximo de 512 threads.
// Como tal, torna-se possível fazer o «unroll» do ciclo for da seguinte forma:
    if (blockDim.x >= 512) {
        if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
        __syncthreads();
    }
    if (blockDim.x >= 256) {
        if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
        __syncthreads();
    }

    if (blockDim.x >= 128) {
        if (tid <  64) { sdata[tid] += sdata[tid +  64]; }
        __syncthreads();
    }

// no unroll do ultimo warp de threads do bloco não há necessidade
// de operações de sincronização pois o hardware garante que a execução
// dos 32 threads do warp é perfeitamente sincronizada.
    if (tid < 32)
    {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
        sdata[tid] += sdata[tid +  1];
    }

    // escrever o resultado do bloco para memória partilhada
    if (tid == 0) c_out[blockIdx.x] = sdata[0];
}
```

Listagem 9 – código da segunda kernel CUDA para a resolução de produtos internos.

Como referido anteriormente, queremos agora evitar ao máximo a utilização do canal PCI-Express, por forma a evitar as latências verificadas na primeira solução conseguida. Para isso, e tendo em conta que a chamada à *kernel* apresentada na listagem anterior (Listagem 9) apenas calcula resultados parciais que devem depois ser somados, deve-se arranjar uma forma de fazer o resto da operação no espaço de memória do GPU.

O CUDA não possui nenhuma primitiva de sincronização global, entre blocos de uma grid, mas como já foi referido antes, o fim de execução de uma *kernel* garante que todos os resultados parciais foram escritos em memória global por todos os blocos da sua grid de execução. Esta propriedade pode ser encarada como uma forma de sincronização global e, apesar de não podermos fazer com que uma única *kernel* faça todo o cálculo com recurso apenas à memória partilhada, podemos recorrer a chamadas consecutivas, do lado do CPU, de uma segunda *kernel* que faça a redução dos resultados parciais.

Para tal, recorreu-se a uma implementação para reduções em CUDA fornecida no SDK da NVIDIA [SDK07]. A *kernel* de redução não é aqui explicada por se encontrar muito bem documentada no mesmo SDK. A solução passa por chamadas sucessivas desta *kernel* de redução até obter o valor final.

Assim, na nossa abordagem, na sua primeira invocação, a *kernel* de redução [SDK07] reduz um vector com os resultados parciais da execução da *kernel* do produto interno. Depois, a cada chamada recursiva, reduz os resultados parciais da sua ultima invocação até lançar um único bloco para a ultima redução.

Os resultados obtidos com a aplicação das alterações à primeira *kernel*, apresentadas na Listagem 9 e com a redução dos resultados feita por completo no GPU são os apresentados na tabela seguinte:

Produto interno	8600M GS (ms)	Speedup	GB/s	GFLOPS	8800 GTS (ms)	Speedup	GB/s	GFLOPS
kernel 2 – 100000 elementos	0,356	NA	1,61	0,56	0,116	2,21	4,93	1,72
kernel 2 – 1000000 elementos	1,8587	1,4	3,08	1,08	0,4305	6,16	13,29	4,65
kernel 2 – 10000000 elementos	15,510599	1,68	3,69	1,29	2,2511	11,63	25,42	8,88

Tabela 9 – tempos de execução médios da segunda kernel CUDA para produtos internos no GPU, com recurso à kernel de redução fornecida com o CUDA SDK [SDK07]. Apresenta ainda o aproveitamento da largura de banda (GB/s) e número de operações de vírgurla flutuante por segundo (GFLOPS).

Estes resultados são mais encorajadores, uma vez que apresentam um speedup médio sobre o tempo de execução da primeira *kernel* de 4.9x para a GeForce 8600 M GS e de 9.3x para a GeForce 8800 GTS, além de que já ultrapassam ligeiramente o tempo de execução em CPU, no caso da GeForce 8800GTS, mesmo no teste mais pequeno. É de notar que estes tempos ainda apresentam o tempo de transferência do valor resultado através do PCI-Express.

A biblioteca CUBLAS fornece um método para produtos internos entre vectores, o cublasSdot(). Este método faz a computação paralela do produto interno entre dois vectores densos de precisão simples e, tal como nos estudos feitos sobre soma e subtracção de vectores densos, foram realizados testes de performance com este método, por forma a comparar a performance obtida com a *kernel* mais rápida implementada. Os resultados obtidos com o cublasSdot() são os apresentados na tabela seguinte:

Produto interno	8600M GS (ms)	GB/s	GFLOPS	8800 GTS (ms)	GB/s	GFLOPS
cublasSdot() – 100000 elementos	0,23652	3,23	0,85	0,085708	8,90	2,33
cublasSdot() – 1000000 elementos	0,98112	7,78	2,04	0,271879	28,06	7,36
cublasSdot() – 10000000 elementos	7,9236	9,63	2,52	1,540232	49,53	12,99

Tabela 10 – tempos de execução médios do método CUBLAS cublasSdot(), aproveitamento da largura de banda (GB/s) e número de operações de vírgurla flutuante por segundo (GFLOPS).

Perante estes resultados, torna-se claro que o cublasSdot é uma melhor solução para este tipo de problemas, comparando os tempos com os das *kernels* desenvolvidas.

Assim, os speedups conseguidos pelo método cublasSdot(), face aos tempos de CPU para os mesmos testes, são os apresentados na tabela seguinte:

Produto interno	C2Duo 2.4GHz	8800 GTS (ms)	speedup
teste 1 – 100000 elementos	0,256	0,085708	2,99
teste 2 – 1000000 elementos	2,6538	0,271879	9,76
teste 3 – 10000000 elementos	26,1786	1,540232	17,00

Tabela 11 – speedups conseguidos com a kernel do CUBLAS para produtos internos no GPU, face aos tempos de CPU para os mesmos testes.

Terminado assim, o estudo sobre as operações sobre vectores que se pretendem utilizar para a paralelização do algoritmo MPCG, torna-se determinante, agora, a escolha das estruturas apropriadas para a implementação de matrizes esparsas no GPU, que ofereçam suporte às necessárias operações de multiplicação entre matrizes esparsas e vectores densos.

5.3 O CNC – Concurrent Number Cruncher

O CNC [Bua07] é uma biblioteca *open source*, originalmente desenvolvida sobre a plataforma CTM [ATI07] (*Close To the Metal*) da ATI e também disponível em CUDA, que recorre ao GPU para acelerar algoritmos de optimização e métodos de integração numérica para sistemas de equações diferenciais parciais. Esta biblioteca é capaz de resolver problemas irregulares de grandes dimensões, recorrendo a um formato genérico para a representação de matrizes esparsas, e especialmente desenhado para um melhor aproveitamento das capacidades do GPU. O CNC implementa o método dos gradientes conjugados com pré-condicionante de Jacobi, recorrendo a uma nova forma para representação de matrizes esparsas, o formato BCRS (Block Compressed Row Storage).

O método dos gradientes conjugados pré-condicionado é um método amplamente utilizado para a resolução iterativa de sistemas definidos positivos de equações lineares. Dado que é um método iterativo, ele pode ser utilizado para resolver sistemas esparsos de grandes dimensões. No entanto, aqui não nos interessa discutir a implementação em [Bua07] deste método, sendo que a sua descrição operacional já foi referida no capítulo anterior. O que nos interessa, de facto, é perceber quais foram as estruturas utilizadas em [Bua07] para a representação de matrizes esparsas e como foi implementado o produto entre matrizes esparsas e vectores.

Para a representação de matrizes esparsas de uma forma genérica, o formato CRS – Compressed Row Storage – é uma estrutura de dados eficiente e amplamente utilizada, guardando num vector apenas os elementos diferentes de zero existentes na matriz, linha a linha. Este formato utiliza endereçamento indirecto, baseado em duas *lookup-tables* para obter os valores (Figura 20).

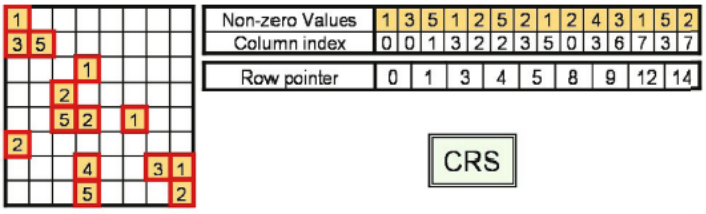

Figura 20 - exemplo do formato CRS para uma matriz esparsa.

Ora, a implementação do método dos gradientes conjugados, qualquer das suas vertentes, envolve operações de multiplicação entre uma matriz muito esparsa e um vector denso, operação que requer um elevado tempo de computação. Este produto, $y \leftarrow \boldsymbol{Ax}$, pode ser expresso na forma:

$$\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n, \quad y[i] = \sum_j a_{i,j} * x_j$$

Tendo em conta que a equação anterior percorre todas as linhas da matriz de forma sequencial, ela pode ser facilmente implementada, para uma matriz *n* x *n* usando o formato CRS, da seguinte forma:

$$\boldsymbol{for}\ i = 0\ \boldsymbol{to}\ n-1\ \boldsymbol{do}$$

$$y[i] = 0$$

$$\boldsymbol{for}\ j = row\_pointer[i]\ \boldsymbol{to}\ row\_pointer[i+1] - 1\ \boldsymbol{do}$$

$$y[i] \leftarrow y[i] + values[j] \times x[column\_index[j]]$$

$$\boldsymbol{end}$$

$$\boldsymbol{end}$$

Na implementação do CNC [Bua07], cada *thread* lançado em paralelo é responsável por determinar o valor de um único índice do vector $\boldsymbol{y}$, na computação da multiplicação da matriz esparsa pelo vector denso. Desta forma, cada *thread* itera através de todos os elementos de uma única linha da matriz esparsa **A** para calcular o produto.

5.3.1 O formato BCRS

O formato de dados BCRS – *Block Compressed Row Storage* – foi desenvolvido em [Bua07] para representar, de uma forma genérica, matrizes esparsas, respeitando uma estrutura ligeiramente diferente do formato CRS, com o objectivo de obter uma estrutura de dados mais adequada ao processamento paralelo em GPU. O BCRS implementado no CNC agrupa os coeficientes não-nulos da matriz em blocos, de dimensões 2×2 ou 4×4, por forma a maximizar o uso da largura de banda da memória gráfica dos GPUs e a reduzir o número de leituras de memória global (Figura 21), o que reduz também a latência.

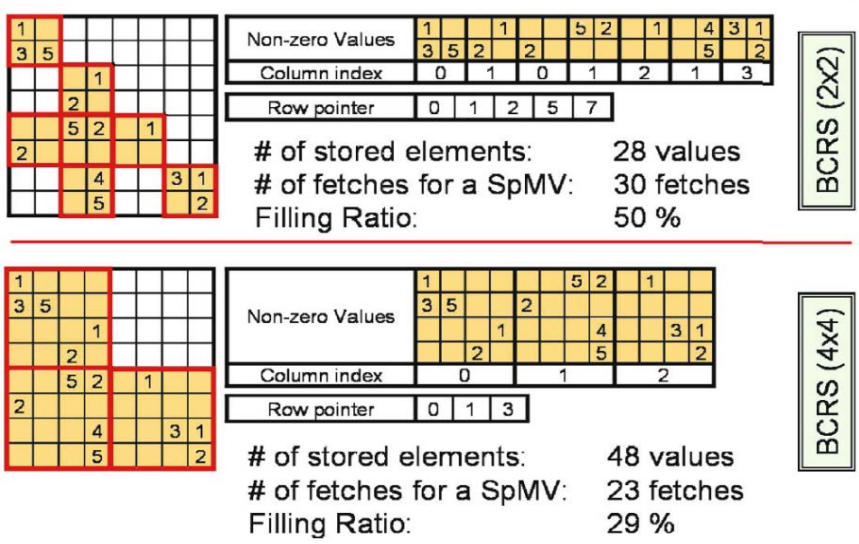

Figura 21 – exemplo do formato BCRS 2x2 e BCRS 4x4 para uma matriz esparsa.

Na figura, o número de elementos guardados (*#of stored elements*) diz respeito a todos os elementos guardados, nulos ou não, coloridos a amarelo na figura. O número de leituras de memória global (*#of fetches*) necessários para realizar um produto entre a matriz e um vector denso são também fornecidos, sendo que menos leituras de memória global implicam menos latência. Um outro dado apresentado na figura é a percentagem média de elementos não-nulos por bloco (*Filling Ratio*). Este dado é importante na medida em que, quanto maior for esta percentagem, mais eficientes são as computações do bloco no GPU.

Quando faz a computação de um bloco BCRS, o GPU lê uma única vez os valores associados ao vector $\boldsymbol{x}$ para todo o bloco e guarda-os em registos para os reutilizar para cada linha do bloco, minimizando o número de leituras de memória global.

O formato BCRS, ao guardar blocos de maior dimensão, reduz o número de blocos e o número de redireccionamentos a serem resolvidos na pesquisa de um bloco, durante a operação de multiplicação. Isto reduz a latência nos acessos a memória, pois cada redireccionamento resulta em acessos a memória global dependentes.

No entanto, é importante considerar que, embora um maior tamanho do bloco seja ideal para reduzir a latência, um bloco de 4x4 é apenas adequado a matrizes mais compactas, por forma a evitar que os blocos tenham muitos valores nulos. De outra forma, um bloco de menores dimensões, 2x2, torna-se mais adequado.

5.4 Integração do CNC no simulador

Como referido no 4º capítulo deste documento, para um sistema com n partículas, os vectores terão $3n$ valores, dadas as componentes x, y e z dos valores da força e da velocidade de uma partícula. Também, por consequência destas componentes, as matrizes esparsas do simulador desenvolvido em [Birr07] são, na realidade, matrizes quadradas de blocos de 3×3 valores, sendo formadas por um vector de listas, com uma lista para cada linha de matriz, sendo que, para cada linha, são guardados os blocos de 3×3 com valores não nulos dessa linha.

Como foi referido na secção anterior, o CNC implementa representações para matrizes esparsas em blocos de 2×2 e 4×4 e, como forma de representação genérica para matrizes esparsas, implementa a representação CRS. Além disto, um outro factor importante de salientar, é que as matrizes do CNC guardam todos os valores não nulos da matriz, ao contrário do que acontece nas matrizes originais do simulador, onde se aproveita a simetria destas matrizes para guardar apenas metade dos valores. Este tema será melhor explicado adiante.

Nesta secção, pretende-se avaliar qual a melhor representação a utilizar para as matrizes esparsas a utilizar na implementação GPU do MPCG. Afinal, como salientado no capítulo anterior, este método iterativo é fundamentalmente dominado por esta operação. Numa primeira fase, foi necessário desenvolver uma forma de converter o formato original das matrizes rarefeitas do simulador para o formato genérico do CNC. Uma vez que as matrizes originais são matrizes de blocos 3×3, formato que não é suportado pelo CNC, estendeu-se posteriormente o CNC por forma a suportar, também, o formato BCRS 3×3, implementando a respectiva operação de multiplicação da matriz por vectores densos em CUDA.

Como referido antes, na implementação em GPU, não tiramos partido do facto de as matrizes serem simétricas, guardando todos os valores não nulos da matriz. Isto deriva do facto de que o GPU da GeForce 8800GTS apenas suporta a versão 1.0 do CUDA, o que significa que este GPU não permite operações atómicas de escrita em memória global, suportadas a partir da versão 1.1 do CUDA [Cud07]. Ao não permitir este tipo de operações, se for programada uma *kernel* que, quando lançada para executar numa *grid*, obrigue mais que um único *thread* a escrever para um mesmo endereço de memória global de forma não atómica, o comportamento é imprevisível. Ora, para implementar em GPU a operação de multiplicação da matriz esparsa pelo vector denso guardando apenas metade dos valores não nulos e aproveitando a simetria existente nas matrizes, obriga a que um mesmo endereço de memória global seja escrito por mais que um *thread*. Assim, por limitação do *hardware* disponível, não foi explorada a possibilidade de utilizar operações atómicas por forma a reduzir os acessos de leitura de memória global. Deste modo, nas operações de multiplicação entre matrizes esparsas e vectores densos suportadas no CNC, as matrizes guardam todos os índices não nulos da matriz, obrigando o GPU a efectuar o dobro dos acessos a memória que a respectiva versão do CPU.

Por forma a fazer uma escolha fundamentada da representação a utilizar na implementação do MPCG, realizou-se, primeiro, um estudo com o objectivo de verificar qual a taxa de ocupação das matrizes utilizadas no simulador. Para simulações onde não exista variação da malha verificou-se que existem, em média, 6 blocos de 3×3 *floats* não nulos por linha. Torna-se assim possível determinar a taxa de ocupação destas matrizes, em função do número de partículas utilizadas na simulação.

Implementou-se, então, um *benchmark* para matrizes com taxas de ocupação semelhantes às utilizadas no simulador, testando o tempo de execução da multiplicação da matriz por um vector denso, com as várias representações disponíveis para a matriz esparsa. O Gráfico 1 apresenta os tempos de execução da operação GPU de multiplicação de uma matriz rarefeita por um vector denso, com as diferentes representações para as matrizes esparsas, e com matrizes de dimensões várias.

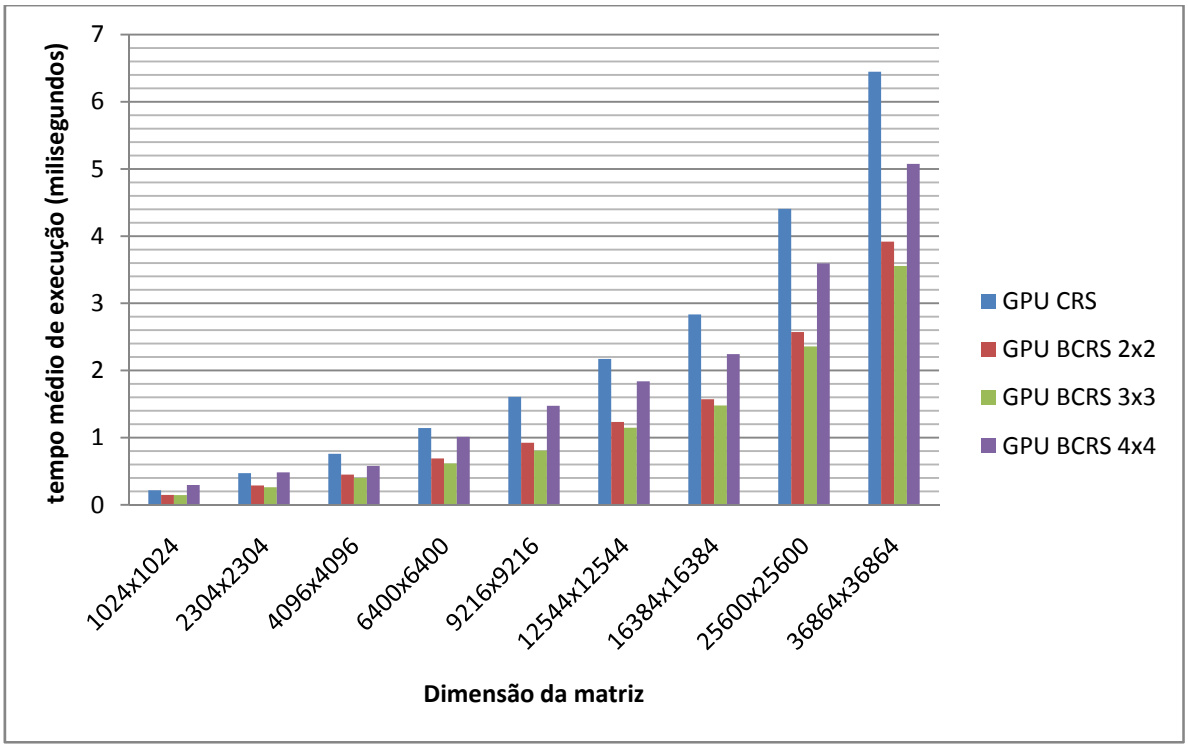

Gráfico 1 – tempos de execução da operação de multiplicação de uma matriz esparsa por um vector denso em GPU – representações CRS (Compressed Row Storage) e BCRS (Block CRS) com blocos de diferentes dimensões.

Como é visível no gráfico anterior, o formato BCRS 3×3 revela-se o mais eficiente para efectuar a computação do produto matriz-vector em GPU. Isto faz todo o sentido, se tivermos em conta o formato das matrizes originais do simulador. Como referido no início desta secção, por consequência das componentes x, y e z dos valores da força e da velocidade de cada partícula, as matrizes rarefeitas originais são matrizes de blocos 3×3. Estes blocos têm quase sempre 100% de valores não nulos e podem, ou não, ser contíguos. Assim, ao fazer a conversão destas matrizes para blocos de outras dimensões, 2×2 ou 4×4, é natural que existam perdas de performance, como se verificam na prática. No caso dos blocos 2×2, não só o número de blocos a processar será necessáriamente maior, como poderão haver blocos com apenas um valor não nulo dos 4 do bloco, como no caso de termos um bloco de 3×3 isolado. No caso dos blocos de 4×4, apesar de teoricamente existir um menor número de blocos para processar, estes poderão ter uma taxa de ocupação de elementos não nulos muito reduzida, o que obriga a computação desnecessária, degradando a performance, como verificado na prática.

Apesar de implicar mais acessos a memória global e de ocupar mais espaço em memória, quando comparado com a performance em CPU, é fácil verificar a superioridade do GPU para fazer este tipo de cálculos, sendo atingido um *speedup* de $6.0 \times$ para o teste mais pequeno e de $16.6 \times$ para o mais pesado (Gráfico 2).

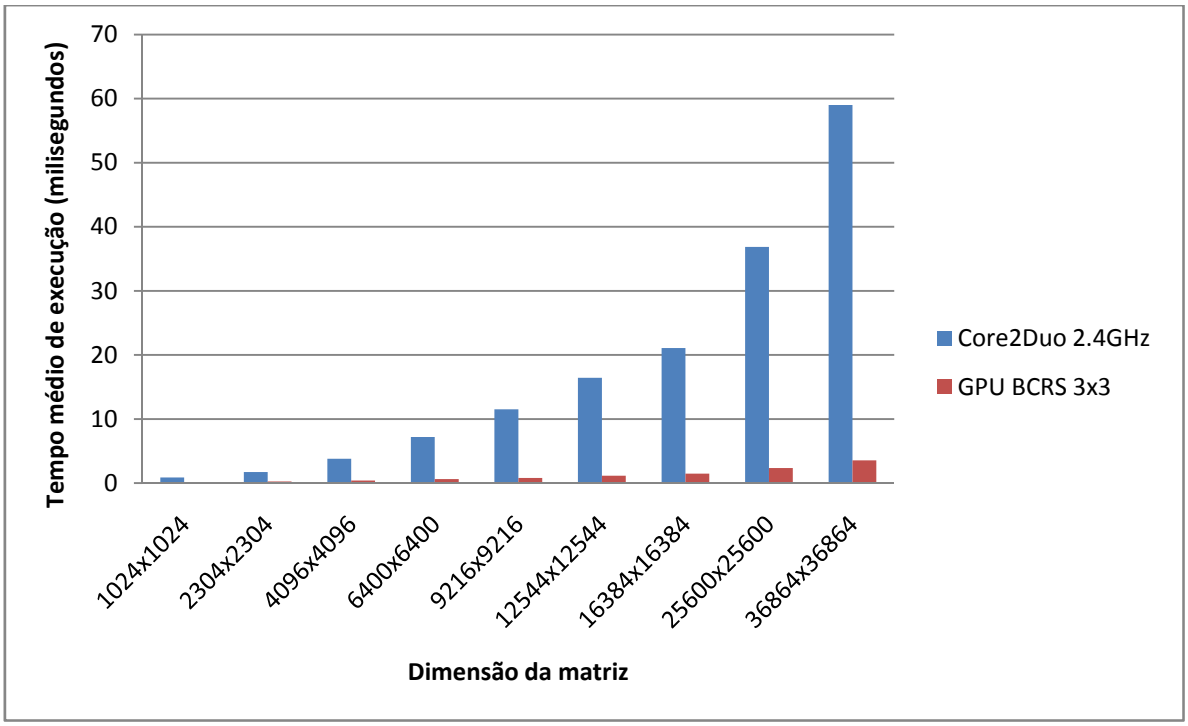

Gráfico 2 – comparação dos tempos de execução da operação de multiplicação de uma matriz esparsa por um vector denso em GPU e CPU.

Todos os testes realizados foram feitos para matrizes rarefeitas com uma taxa de ocupação muito reduzida, sendo esta taxa calculada de acordo com as propriedades demonstradas pelo simulador, desenvolvido em [Birr07], para malhas sem variação dinâmica do nível de detalhe. Neste tipo de malhas, apenas existem 6 interacções por partícula, o que significa que, independentemente do tamanho da matriz, cada linha tem exactamente 6 blocos de 3×3 valores não nulos.

5.5 O MPCG GPU Solver

A implementação do MPCG em GPU, insere-se no trabalho desenvolvido em [Birr07]. Internamente, o simulador desenvolvido em [Birr07] encontra-se dividido em diversas componentes, realizadas sob a forma de bibliotecas, tão autónomas quanto possível. Para além de uma biblioteca nuclear, que contém as classes abstractas dos componentes base do sistema como partículas, estruturas, fontes, interacções e campos, podem também encontrar-se as bibliotecas relativas à detecção de colisões, aos métodos de integração numérica, às operações envolvendo matrizes esparsas e às classes que implementam o modelo de tecidos (Figura 22), estando acentuada a amarelo a biblioteca relativa aos métodos de integração numérica.

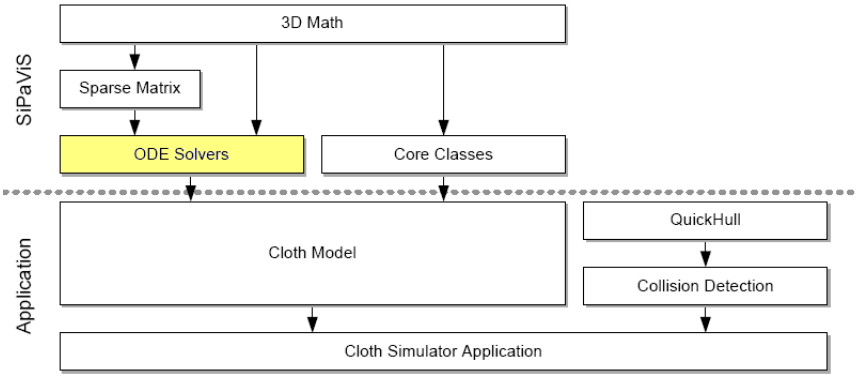

Figura 22 - Constituintes internos do simulador desenvolvido em [Birr07].

O trabalho desenvolvido nesta dissertação consistiu, fundamentalmente, numa extensão à biblioteca relativa aos métodos de integração numérica, da forma mais independente possível de todas as outras componentes, com o *solver* para o método dos gradientes conjugados, apresentado no 4º capítulo deste documento, em GPU.

Quando dizemos que o trabalho foi desenvolvido da forma mais independente possível, é porque os métodos de integração numérica, como é visível na figura, utilizam objectos das bibliotecas de matemática e de suporte a operações sobre matrizes esparsas e, como tal, qualquer alteração a estas bibliotecas de base, terá implicações sobre o trabalho desenvolvido. Para uma descrição promenorizada sobre os vários componentes da figura (Figura 22), por se encontrar fora do âmbito deste trabalho, recomenda-se a leitura de [Birr07].

Como referido antes, o GPU comunica com o CPU através do bus PCI-Express (Figura 23). O PCI-Express 16x é capaz de alcançar 4GB/s na transferência de dados em cada sentido, logo, o pico máximo de largura de banda na comunicação com o CPU é de 8 GB/s, enquanto o pico máximo da largura de banda da memória gráfica é, pelo menos, uma ordem de magnitude, ou várias, acima da largura de banda do PCI-Express. Isto pode provocar sérios *bottlenecks* se a intensidade aritmética dos cálculos efectuados no espaço de memória do GPU, entre transferências de dados no PCI-Express, não for suficientemente pesada para disfarçar a latência na transferência de dados através deste bus de comunicação.

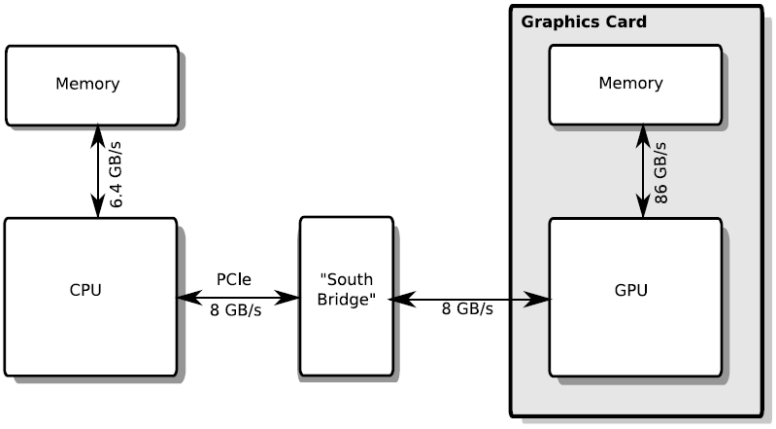

Figura 23 - Esquema típico de um sistema que use o GPU como um co-processador.

A plataforma CUDA permite-nos um ambiente de programação heterogéneo onde é possível misturar código paralelo (GPU), com código sequencial (CPU), num mesmo programa. O nosso *solver* GPU tira partido desta possibilidade.

O *solver* existe no espaço de memória do CPU, sendo que este é detentor do controlo da execução durante todo o processo. Ao início, todas as estruturas necessárias à resolução do método dos gradientes conjugados, apresentado no 4º capítulo deste documento, são alocadas e afectadas no espaço de memória global do GPU. Torna-se, assim, possível tirar partido de *kernels* de BLAS para todas as operações sobre vectores envolvidas no método, sem que haja a necessidade de transferência de dados através do PCI-Express de volta para o espaço de memória do CPU, exceptuando nas operações de produtos internos entre vectores, onde o valor resultado é necessário do lado do CPU, por questões de controlo de execução (condição de paragem, etc..). Simultaneamente, as operações de multiplicação entre matrizes esparsas e vectores densos também poderão ser inteiramente realizadas no espaço de memória do GPU, sem a necessidade de transferência de dados através do PCI-Express, sendo que, apenas quando termina a execução do método de integração, se transferem os vectores com os resultados de volta para o espaço de memória do CPU. Com esta abordagem, são minimizadas as transferências de dados pelo PCI-Express, por forma a disfarçar ao máximo a latência destas transferências com computação.

5.5.1 Estruturas de dados utilizadas

As estruturas que representam os vectores do lado do CPU, podem ser vistas como vectores de n elementos, onde n é o número de partículas no sistema. No entanto, cada elemento nestes vectores tem 3 componentes, *x*, *y* e *z*, que dizem respeito às componentes 3D das posições, velocidades, etc, de cada partícula. Assim, cada vector, do lado do GPU, pode ser visto como um vector de $3n$ posições, sendo feito um mapeamento directo entre uma representação e outra.

No caso da representação das matrizes esparsas, do lado do CPU, estas são matrizes de $n \times n$ entradas, onde cada uma das posições (i, j) é ocupada por uma matriz de 3×3 valores. O mapeamento destas matrizes para o espaço de memória do GPU já não é tão trivial. Numa primeira fase, estas são convertidas para a representação CRS (Figura 20), apresentando a matriz rarefeita dimensões $3n \times 3n$. Esta matriz é posteriormente transformada para o formato BCRS, o qual é carregado para o espaço de memória do GPU. Esta opção foi tomada para permitir, de uma forma genérica, escolher qual a representação BCRS – BCRS2x2 ou BCRS4x4 – que se deseja para as matrizes.

O formato BCRS para matrizes (Figura 21) usa 3 tabelas para guardar os índices de colunas, os apontadores para as linhas e os elementos não nulos da matriz, respectivamente. As tabelas que dizem respeito aos apontadores para as linhas ou aos índices de colunas são representados como vectores simples. Para os valores não nulos da matriz, por sua vez, a representação já depende do tamanho de bloco escolhido.

Para os blocos de 2x2, os dados são colocados, bloco a bloco, num vector de blocos 2x2, sendo um bloco de 2x2 representado por um *float4*. Já no caso dos blocos 4x4, os dados são distribuídos em 4 sub-vectores preenchidos com sub-blocos de 2x2 do bloco 4x4 original (Figura 24). Assim, aceder aos 16 valores de um bloco pode ser conseguido com apenas 4 leituras de memória global dos 4 sub-vectores num mesmo índice. Nos blocos de 3x3, a abordagem é semelhante à tomada originalmente no CNC para matrizes de 4x4. Os dados são distribuídos em 3 sub-vectores de *float3*.

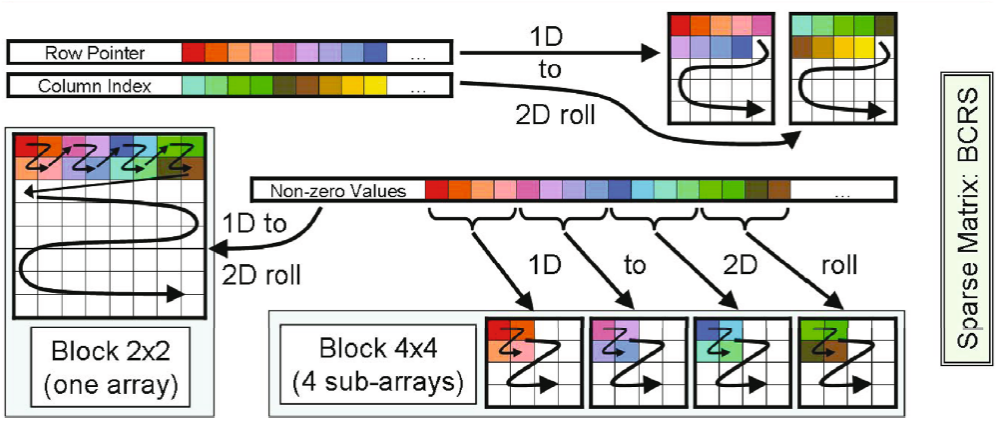

Figura 24 - Representação das matrizes esparsas BCRS no espaço de memória do GPU, retirado de [Bua07].

Como foi referido antes, o trabalho em [Bua07], foi desenvolvido sobre a plataforma CTM e os mapeamentos 1D para 2D, representados na imagem acima para os apontadores de linhas e colunas, dizem respeito a essa implementação, não sendo utilizados na versão em CUDA.

5.5.2 Operações BLAS implementadas

Tendo em conta os estudos realizados, apresentados ao longo deste capítulo, sobre as operações BLAS necessárias à implementação do método dos gradientes conjugados modificado em CUDA, o nosso *solver*, por defeito, recorre ao CUBLAS para as operações sobre vectores e ao método de multiplicação de matrizes esparsas por vectores do CNC, com a versão extendida com matrizes BCRS3x3. Tendo em conta as melhores capacidades do CPU, face ao GPU, para a resolução de sistemas de pequenas dimensões, o nosso solver adapta-se, escolhendo a utilização do CPU ou do GPU em função do número de partículas no sistema.

Também pelo facto de se verificar que a utilização do *solver* recorrendo exclusivamente às *kernels* da biblioteca CUBLAS apresenta melhores resultados, para as operações de BLAS sobre vectores densos, existe a hipótese de escolha, por parte do programador, da utilização dos métodos do CUBLAS em todas as operações sobre vectores ou, alternativamente, da utilização das *kernels* BLAS desenvolvidas na implementação desta dissertação.

6. Resultados

O trabalho desenvolvido nesta dissertação visa a aceleração do trabalho implementado em [Birr07], recorrendo ao modelo de hardware e de programação fornecido pelo CUDA. Tal como foi referido no primeiro capítulo, e aqui repetido por conveniência, a aplicação desenvolvida em [Birr07] pode dividir-se em três fases principais:

- A avaliação do modelo de tecidos (*Evaluation*), onde são determinadas as resultantes das forças, internas e externas, sobre cada partícula do sistema.
- A execução do passo de integração (*Solve*), que inclui a construção das matrizes das derivadas parciais e onde é resolvido o sistema de equações lineares gerado pela avaliação do modelo implícito.
- A variação do nível do detalhe (*DLOD*), que inclui a aplicação da técnica de variação do nível de detalhe, bem como as tarefas de gestão das estruturas de dados associadas aos objectos que vão sendo criados, destruídos, activados ou desactivados, no decorrer da simulação. Note-se que, não havendo variação do nível de detalhe, este conjunto de tarefas é desnecessário, pois não há mudança alguma no conjunto de objectos usados durante toda a execução.

De entre estas 3 fases principais, foi identificada a fase *Solve* como sendo a fase que domina a quase totalidade do tempo de processamento. Nesta fase é resolvido o sistema de equações lineares gerado pela avaliação do modelo implícito através do método MPCG apresentado no 4º capítulo deste documento. As restantes fases, da avaliação do modelo e de variação do nível do detalhe, são consideravelmente inferiores ao peso computacional envolvido na resolução do método de integração e, como tal, não são consideradas aqui, pelo simples facto de que quaisquer optimizações nelas realizadas traduzir-se-iam, provavelmente, em ganhos finais menos notórios, por terem, à partida, menos margem para optimização.

Procedeu-se, como apresentado nos 4º e 5º capítulos, à implementação do MPCG recorrendo a programação genérica em GPU, através da plataforma de desenvolvimento CUDA da NVIDIA, que foi apresentada em maior pormenor no 3º capítulo. Nas secções seguintes, apresentam-se e discutem-se os resultados obtidos com esta abordagem. Os resultados apresentados neste capítulo dizem todos respeito a uma mesma simulação, que corresponde ao cair de um pano, preso por duas pontas, sobre uma esfera, como ilustrado na sequência de imagens da Figura 25.

Para cada teste, fez-se variar o número de partículas que compõem a malha, ajustando as constantes físicas por forma a tratar-se da mesma peça de tecido, com a mesma massa total e o mesmo comportamento dinâmico.

Através da implementação do MPCG em GPU, constatou-se que os ganhos conseguidos nos tempos de execução, para a aceleração global do passo *Solve* do processo de simulação desenvolvido em [Birr07], foram relativamente pequenos, como apresentado no gráfico da Figura 26 abaixo.

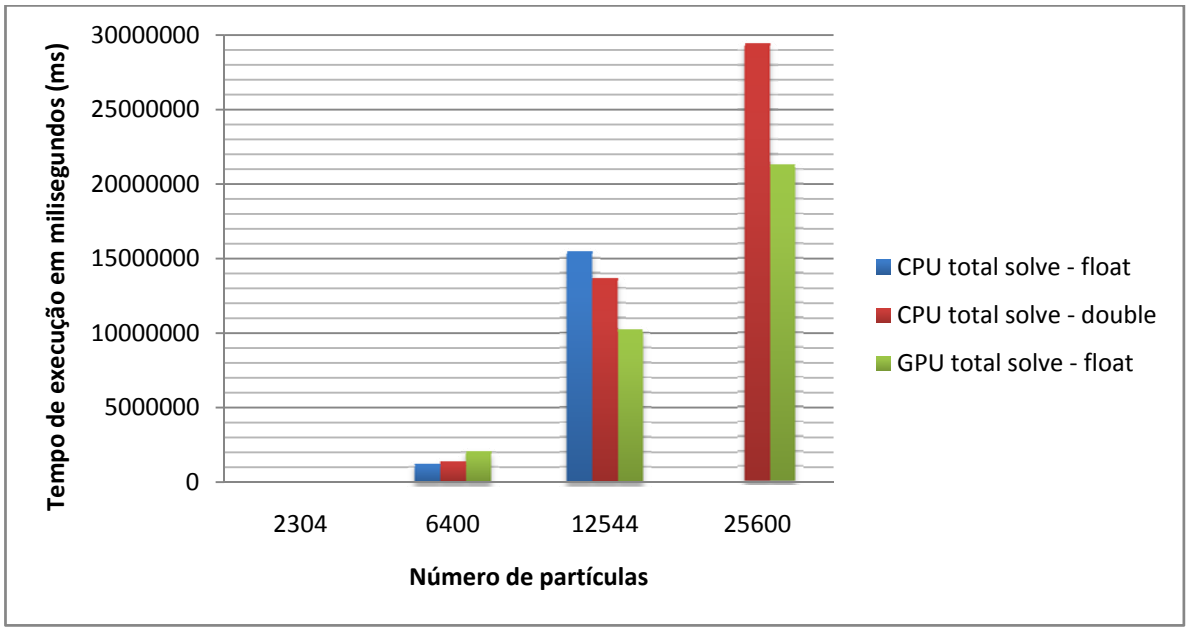

Figura 25 - Tempo total de execução do passo Solve para a simulação (ver Figura 23), para malhas com diferentes números de partículas.

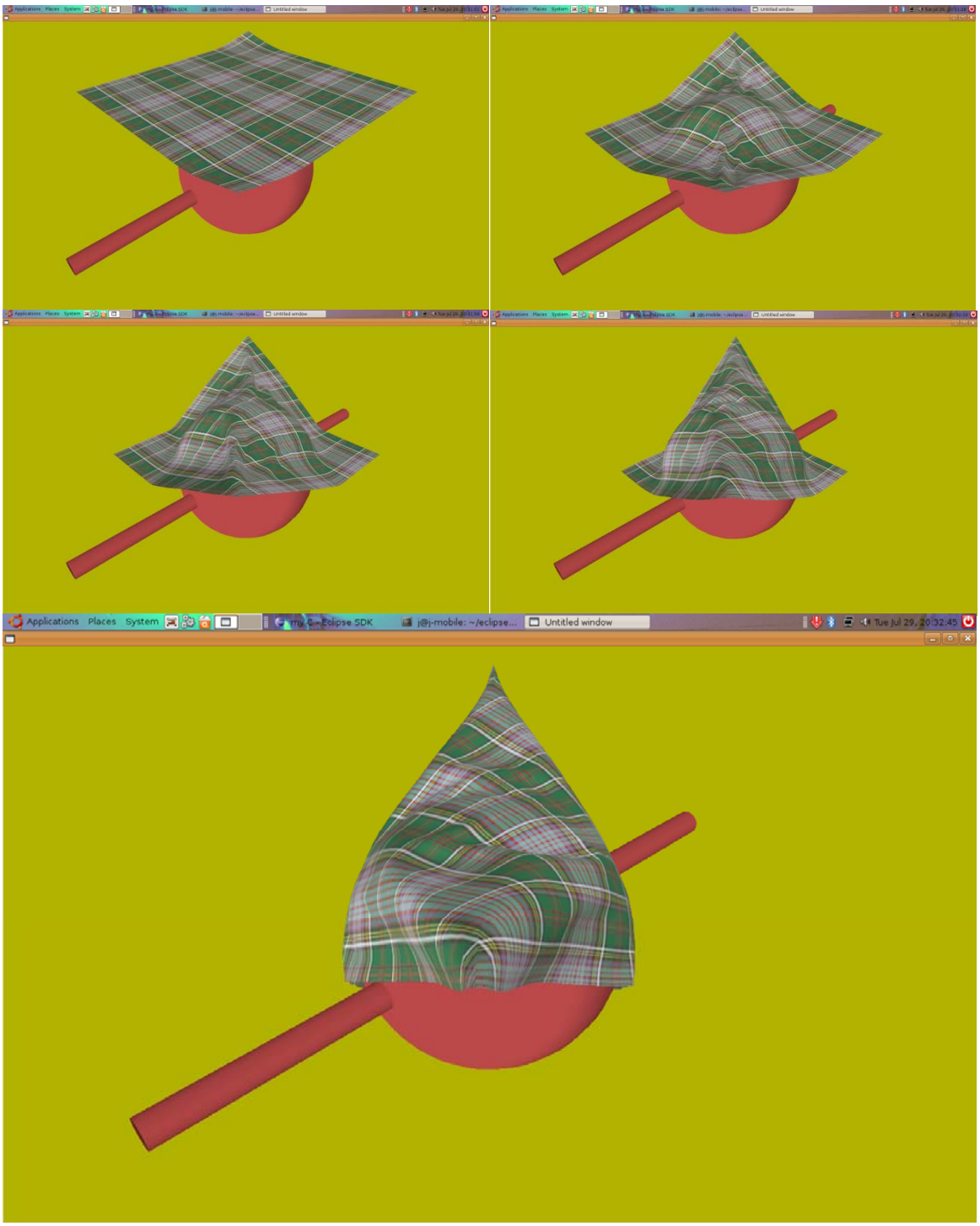

Figura 26 - sequência de imagens ilustrando a simulação feita para os vários testes efectuados para a apresentação de resultados.

Desde logo, o facto de não ser possível a utilização de números de precisão dupla no GPU, ao contrário do que acontece em CPU, tem implicações no número de iterações que o MPCG necessita para convergir, sendo necessário um maior número de iterações consoante o número de partículas aumenta. Para os testes apresentados no gráfico, enquanto o CPU, recorrendo a precisão dupla, consegue manter um máximo de 100 iterações em todos os testes, sendo *n* o número de partículas do sistema, no caso do GPU apenas são necessárias 100 iterações no teste onde $n = 6400$, sendo necessárias 150 iterações para convergência do MPCG quando $n = 12544$ e um total de 200 iterações para $n = 25600$.

A necessidade deste maior número de iterações para a convergência do MPCG no GPU, está relacionada com a falta de precisão nos cálculos efectuados pelo GPU. Esta falta de precisão é «potenciada», sobretudo, pelas operações de produto matriz-vector e de produto interno entre vectores. Estas operações envolvem muitas operações de multiplicação + soma (MADs – *Multiply and Add operations*) e este tipo de operações são «truncadas» pelo *hardware*, por forma a efectuar as duas operações num único *clock*, o que se reflete nesta perda de precisão.

O uso de floats no CPU também afecta o número de iterações necessárias para a convergência do algoritmo, o que acaba por ser visível também no gráfico. Além disso, implica que todas as operações da simulação são realizadas com precisão simples o que afecta largamente a precisão de toda a simulação. Estes efeitos sobre a precisão reflectem-se na necessidade de aumentar o número de iterações no CPU para 200 na simulação com 12544 partículas e tornando-se impraticavel para a simulação com 25600 partículas, sendo que nem com um máximo de 1500 iterações o MCPG converge.

É, também, de referir que a fase *solve* do processo de simulação, identificada como a mais pesada das 3 fases que compõe o algoritmo, não é apenas composta pela resolução do sistema de equações diferenciais através do MPCG, incluindo também a construção das matrizes das derivadas parciais, processo que não foi alvo de aceleração.

Devemos ainda considerar que todo o *overhead* associado à conversão de estruturas e à cópia dos dados para o espaço de memória do GPU, para a realização do MPCG, está incluído nos tempos GPU expressos acima.

Se observarmos a Figura 27, que diz respeito aos tempos de execução do método MPCG, por iteração, os resultados são notoriamente mais satisfatórios. Nestes tempos, estão já retirados quaisquer *overheads* relacionados com conversões de dados e tranferências de dados entre CPU e GPU.

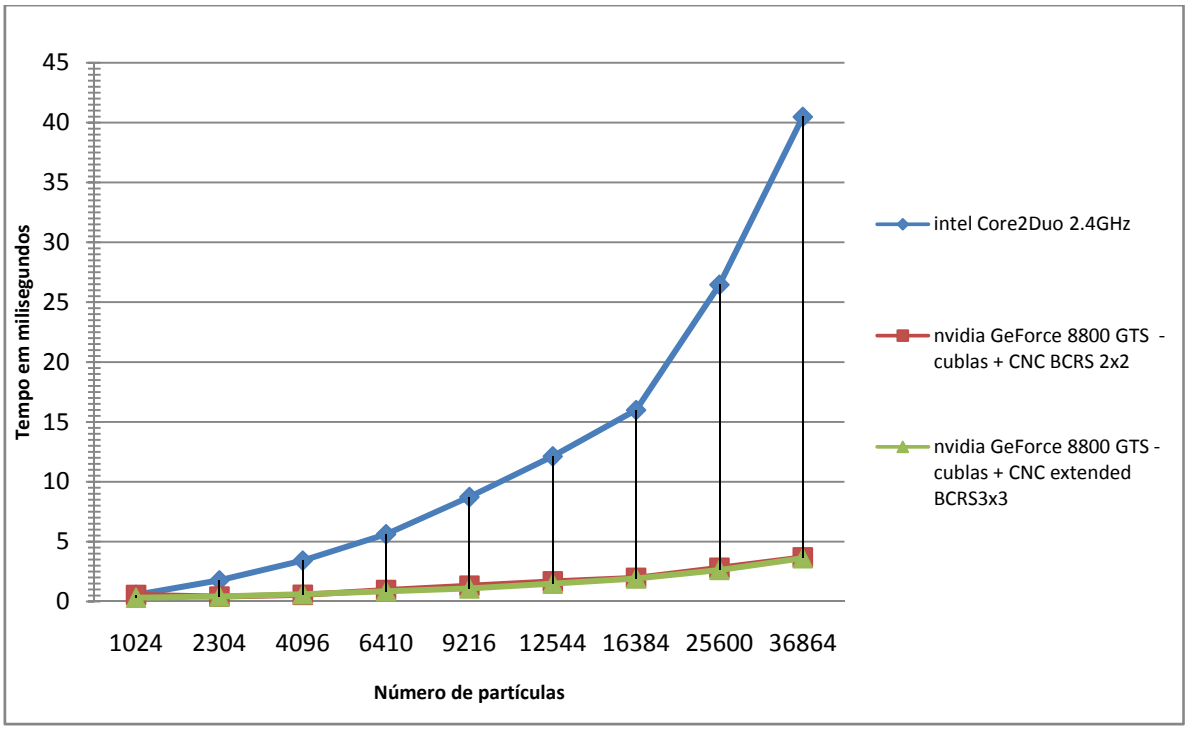

Figura 27 - Evolução do tempo médio, por iteração, na execução do MPCG variando o número de partículas no sistema.

Na verdade, ao considerarmos apenas o tempo de execução do MPCG, o GPU apresenta um crescimento do tempo médio por iteração quase linear e de acentuação reduzida, em contraste com a evolução deste mesmo tempo no CPU, de aspecto praticamente exponencial. Obtém-se assim um speedup de aproximadamente 11 × no teste mais pesado que nos foi possível realizar, com 36864 partículas (Figura 28). Tendo em conta os resultados obtidos para as operações individuais a que recorre o MPCG quando operados sobre grandes volumes de dados, apresentados no 5º capítulo deste documento, teria sido interessante testar sistemas com maior número de partículas, cenário mais adequado às *kernels* desenvolvidas e apresentadas nesse mesmo capítulo. Isto não foi possível por limitação do *hardware* disponível.

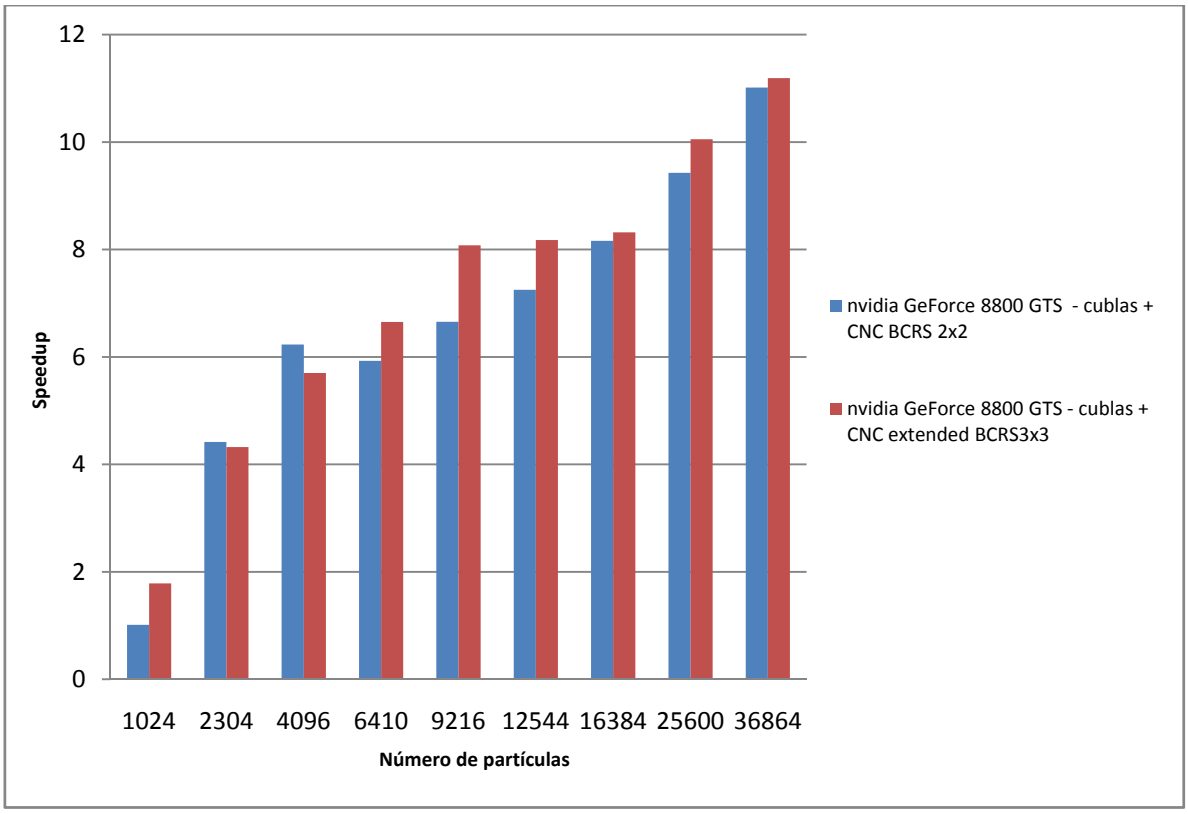

Figura 28 - Speedups obtidos face ao Core2Duo 2.4GHz, em função do número de partículas. São também visíveis as diferenças entre o speedup obtido recorrendo ao formato original BCRS2x2 do CNC e o speedup obtidos com a versão extendida, suportando BCRS 3x3.

7. Conclusões e trabalho futuro

A motivação principal que serviu de impulso para o início dos trabalhos desta dissertação foi a ideia de acelerar um simulador de tecidos com realismo acrescido [Birr07] tirando partido da capacidade computacional do GPU moderno. O GPU moderno pode ser encarado como um co-processador programável, para processamento paralelo genérico, com uma enorme capacidade de computação, e revela-se especialmente adequado para a implementação de operações paralelas sobre grandes volumes de dados, como é visível nos resultados obtidos nos 5º e 6º capítulos deste documento. Assim, pode-se encarar a programação genérica em GPU como uma boa abordagem para a aceleração de algoritmos computacionalmente muito exigentes, através da identificação de fases cujas operações sejam paralelizáveis, segundo o modelo SIMD do *hardware* do GPU. Outra das motivações deste trabalho foi a exploração da plataforma da NVIDIA, o CUDA, que consiste num modelo de *hardware* e de programação para GPU inovador e que esconde algumas da limitações da programação em GPU por APIs gráficas, apresentando o GPU como um verdadeiro co-processador genérico.

A plataforma CUDA simplifica muito a aprendizagem da programação em GPU, bem como permite retirar todo o poder computacional do GPU, ao expor o *hardware* do GPU de uma forma mais transparente. O facto do seu modelo de programação consistir numa extensão à linguagem C, familiar a um grande leque de programadores, torna a curva de aprendizagem de programação em GPU bastante mais ténue face às alternativas de programação com recurso a linguagens de *shading*. Além disto, apresenta também a enorme vantagem de permitir programar o GPU de uma forma mais genérica e flexível, não havendo a necessidade de tratar os problemas como problemas de píxeis, vértices ou geometria. É ainda de referir que, tendo em conta os preços relativamente acessíveis dos GPUs modernos e a capacidade computacional revelada, o GPU moderno pode ser encarado como uma ferramenta para processamento paralelo extremamente barata e competitiva.

Tal como relembrado no início do 6º capítulo do documento, a fase de simulação que se pretendeu acelerar foi a fase *Solve*, que se divide em duas «sub-fases» distintas:

- a construção das matrizes das derivadas parciais
- a resolução do sistema de equações lineares gerado pela avaliação do modelo implícito, através do método iterativo MPCG.

Tal como apresentado nesse mesmo capítulo, a aceleração conseguida para o total desta fase foi relativamente pequena, tendo sido obtido um speedup absoluto de apenas 1.4x para um sistema com 25600 partículas. No entanto, se tivermos em conta a Figura 29, onde é apontado com uma seta, o pico máximo aproximado, em termos de GFLOPS, do *hardware* utilizado neste trabalho, podemos esperar resultados bastante melhores para GPUs mais recentes.

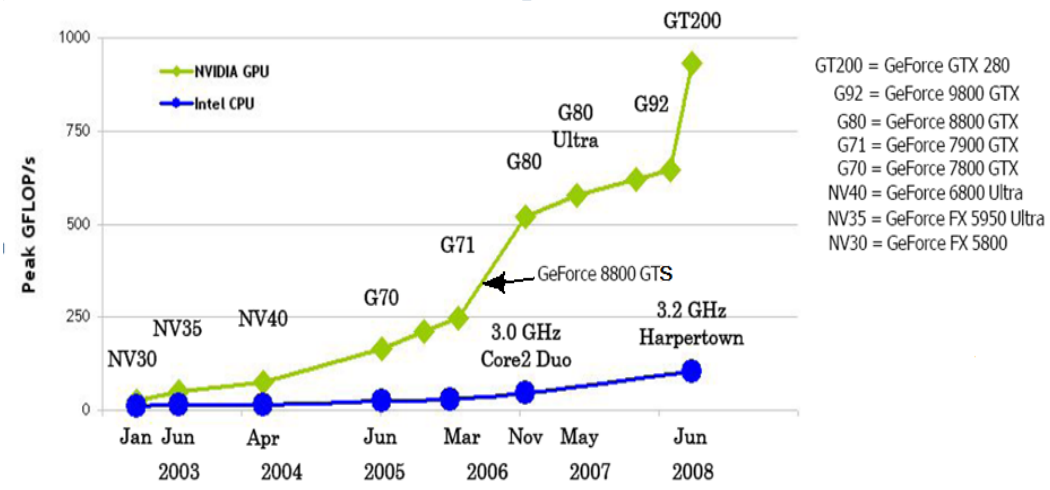

Figura 29 - evolução da capacidade de processamento do GPU face à mesma evolução do CPU, entre Janeiro de 2003 e Junho de 2008, sendo apontado o pico máximo de performance, aproximado, da nvidia GeForce 8800 GTS Fatal1ty, GPU utilizado neste trabalho.

Existem inúmeras abordagens que podem ainda ser tomadas com vista à aceleração do simulador desenvolvido em [Birr07]. Em primeiro lugar, tal como referido antes, apenas o método iterativo existente na fase *Solve* do simulador foi alvo de aceleração nesta dissertação. Uma primeira abordagem poderia passar por acelerar também a construção das matrizes das derivadas parciais. Para tal, seria necessário, por exemplo, estender o CNC para que este suportasse operações de soma e subtracção de matrizes rarefeitas. Por outro lado, poder-se-ia estender também o CNC para suportar operações de multiplicação para matrizes simétricas, aproveitando esta propriedade para guardar apenas metade dos valores em memória, como acontece do lado do CPU. Isto reflectir-se-ia provavelmente em ganho de performance pois implica aproximadamente metade dos acessos de leitura de memória global, perdendo algum paralelismo na escrita em memória global, tirando partido de operações atómicas mas, para tal, seria necessário *hardware* com suporte para CUDA 1.1,

no mínimo. Seria também interessante experimentar *hardware* com suporte para CUDA 2.0 ou superior, o que permitiria a optimização anterior, bem como, alterando ligeiramente o código para que este use *doubles* em vez de *floats*, minimizar o número de iterações necessárias para a convergência do método iterativo.

Uma outra abordagem interessante seria a de analisar a possibilidade de utilizar múltiplos GPUs, procurando porções do próprio código paralelo que sejam independentes entre elas, por forma a distribuir estas porções para serem executadas em paralelo em GPUs diferentes, ganhando, assim, mais um nível de paralelismo.

Poder-se-ia ainda reorganizar o próprio código do MPCG, implementando *kernels* mais complexas, que realizassem porções maiores do algoritmo sem intervenção do CPU, e procurando tirar partido da memória partilhada de alta velocidade. Observando a Listagem 2, um bom exemplo seria uma *kernel* que efectuasse a sequência de operações envolvidas nas linhas 9, 10 e 11 da listagem. Na computação da linha 9, o vector $\propto q$ poderia ser carregado em memória partilhada, evitando a sua leitura de memória global na operação da linha 10, cujo resultado poderia ser computado também para memória partilhada, para evitar também a sua leitura de memória global, podendo ser utilizado directamente na multiplicação pela matriz esparsa da linha 11. Esta abordagem reflectir-se-ia, provavelmente, em algum ganho de performance, pois permitiria disfarçar melhor a latência nos acessos a memória.

Para acelerar o simulador, no limite, o ideal seria colocar toda a simulação no GPU, evitando as constantes trocas de informação entre o CPU e o GPU.

Um outro motivo de trabalho futuro poderia, ainda, ser a realização de trabalhos que visem a aceleração de outras aplicações, computacionalmente muito exigentes, com recurso à plataforma CUDA. A flexibilidade de programação e o potencial revelados por esta plataforma associados ao baixo custo, face ao poder computacional, do *hardware* que a suporta, tornam promissor a sua utilização neste tipo de aplicações, tornando-as mais rápidas a um baixo custo.

8. Bibliografia

[Wit95] Andrew Witkin and David Baraff, *Introduction to Physically Based Modelling*. Course Notes, SIGGRAPH 95.

[Asc03] Ascher, U. M., and Boxerman, E. *On the modified preconditioned conjugate gradient method in cloth simulation.* The Visual Computer 19, 7-8(dec 2003), 526-531.

[Asp08] http://www.gpgpu.org/asplos2008/.

[ATI07] *ATI CTM, Close To the Metal – technical reference manual,* 2007, http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[Bar94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst. *Templates for the solution of Linear Systems: Building Blocks for Iterative Methods*, 1994.

[Birr07] Fernando Birra, *Técnicas Eficientes de Simulação de Tecidos com Realismo Acrescido.* Dissertação para a obtenção do grau de Doutor em Informátca pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Lisboa, 2007.

[BLAS] *BLAS – Basic Linear Algebra Subprograms.* http://www.netlib.org/blas.

[Bre92] Breen, D. E., D. H., and Getto, P. H. *A physically-based particle model of woven cloth.* The Visual Computer 8, 5-6 (June 1992), 246-277.

[Bre00] Breen, D. E., *A survey of cloth modeling methods.* In *Cloth Modeling and Animation*, D. H. House and D. E. Breen, Eds A K Peters, Natck, Massachussets, 2000, pp. 19-54.

[Bua07] Luc Buatois, Guillaume Caumon, and Bruno Lévy, *Concurrent Number Cruncher – An Efficient Sparse Matrix Solver on the GPU*. Gocad Research Group, INRIA, Nancy Université, France, 2007.

[Cub07] NVIDIA CUBLAS – *Cuda Basic Linear Algebra Subprograms* – 2007. http://developer.download.nvidia.com/compute/cuda/1_1/CUBLAS_Library_1.1.pdf.

[Cud07] *NVIDIA CUDA, Compute Unified Device Arquitecture – programming guide*, 2007, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.

[Zel05] Cyril Zeller, *Cloth*, NVIDIA SDK9 White Paper, July 2005, http://developer.nvidia.com/object/sdk-9.html.

[Zel07] Cyril Zeller, T. S., *Cloth Simulation*, NVIDIA SDK10 White Paper, February 2007, http://developer.nvidia.com/object/sdk_home.html.

[Bar98] David Baraff and Andrew Witkin, *Large Steps in Cloth Simulation*, Computer Graphics Proceedings (SIGGRAPH 98), Carnegie Mellon University, Orlando, July 1998.

[Bar01] David Baraff, *Physically Based Modeling, Implicit Methods for Differencial Equations*. Course Notes, SIGGGRAPH 2001.

[Nyl07] Lars Nyland, Mark Harris and Jan Prins, *Fast N-Body SimulationWith CUDA*. GPU Gems 3, July 2007, 677-695.

[Hwu07] Wen-Mei Hwu, David Kirk, ECE 498 AL1: *Programming Massively Parallel Processors*, 2007. http://courses.ece.uiuc.edu/ece498/al1.

[Kes07] Kessenich et al. *The OpenGL Shading Language*, 2007.

[Den06] Kjartan Dencker, Torbjørn Hallgren, *Cloth Modelling on the GPU*. Master of Science in Computer Science Report, Norwegian University of Science and Technology, June 2006.

[Mas99] Berna L. Massingill. *Patterns for Parallel Application Programs*, 1999.

[Gov07] Naga K. Govindaraju, Bingsheng He, Burton Smith and Qiong Luo. *Efficient Gather and Scatter Operations on Graphics Processors*, 2007.

[Pro95] Xavier Provot, *Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behaviour*, Institut National de Recherche en Informatic et Automatique (INRIA), B. P. 105, 78153 Le Chesnay Cedex, France, 1995.

[SDK07] NVIDIA CUDA SDK 1.1, http://www.nvidia.com/object/cuda_get.html.

[SIMD] *SIMD Architectures. http://carbon.cudenver.edu/~galaghba/simd.html.*

[She94] ShewChuk, J. R. *An introduction to the conjugate gradient method without the agonizing pain.* Computer Science Tech. Report 94-125,Carnegie Mellon University, Pittsburgh, PA, 1994.

[Ter87] Terzopoulos, D., Platt, J. Barr, A., and Fleisher, K., *Elastically deformable models*. Computer Graphics (Proc. Sigggraph) 21, 4 (July 1987), 205-214.

Anexo A – Operações BLAS implementadas em CUDA

```
float j_cuda_dot(float* vec1, float* vec2, float* vec3, unsigned int size){

    unsigned int numBlocks = (size/(THREADS_VEC_OPS*4))+1;
    dim3 gridSize(numBlocks, 1, 1);
    dim3 blockSize((THREADS_VEC_OPS), 1, 1);
    int smemSize = (THREADS_VEC_OPS)*sizeof(float);

    j_vec_dot<<<gridSize, blockSize, smemSize>>>(vec1, vec2, vec3, size);
    cudaThreadSynchronize();

    int theSize=size;
    int maxThreads = 128;
    int whichKernel = 6;
    int maxBlocks = 64;
    bool cpuFinalReduction = false;
    int cpuFinalThreshold = 1;
    int nBlocks = 0;
    int nThreads = 0;
    getNumBlocksAndThreads2(whichKernel, theSize, maxBlocks, maxThreads, nBlocks, nThreads);
    if (nBlocks == 1) cpuFinalThreshold = 1;
    float* h_odata = (float*) malloc(nBlocks*sizeof(float));
    float* d_odata = NULL;
    cudaMalloc((void**) &d_odata, nBlocks*sizeof(float));
    cudaMemcpy((void**)d_odata, (void**)vec3, nBlocks*sizeof(float), cudaMemcpyDeviceToDevice);

    float gpu_result = 0.0f;

    gpu_result = benchmarkReduce<float>(theSize, nThreads, nBlocks, maxThreads, maxBlocks,
                                        whichKernel, 1, cpuFinalReduction,
                                        cpuFinalThreshold, false,
                                        h_odata, vec3, d_odata);
    cudaFree(d_odata);
    free(h_odata);
    return gpu_result;
}

void getNumBlocksAndThreads(int n, int maxThreads, int &blocks, int &threads)
{
    if (n == 1)
        threads = 1;
    else
        threads = (n < maxThreads*2) ? n / 2 : maxThreads;
    blocks = (n / (threads * 2));
}

void j_cuda_vec_add(float* vec1, float* vec2, float* res, unsigned int size){
        int thds = THREADS_VEC_OPS;
        int blcks = (size/(THREADS_VEC_OPS*4))+1;
        dim3 gridSize(blcks, 1, 1);
        dim3 blockSize(thds, 1, 1);

        J_vec_sum<<<gridSize, blockSize>>>(vec1, vec2, res, size);
}

void j_cuda_vec_scale(float* vec1, float alfa, unsigned int size){
        int thds = THREADS_VEC_OPS;
        int blcks = (size/(THREADS_VEC_OPS*4))+1;
        dim3 gridSize(blcks, 1, 1);
        dim3 blockSize(thds, 1, 1);
        J_vec_scale<<<gridSize, blockSize>>>(vec1, alfa, size);
}
```

```
void j_cuda_vec_sub(float* vec1, float* vec2, float* res, unsigned int size){
	int thds = THREADS_VEC_OPS;
	int blcks = (size/(THREADS_VEC_OPS*4))+1;
	dim3 gridSize(blcks, 1, 1);
	dim3 blockSize(thds, 1, 1);
	J_vec_sub<<<gridSize, blockSize>>>(vec1, vec2, res, size);
}

// sum kernel
__global__ void J_vec_sum(float* vec1, float* vec2, float* res, unsigned int size){

	int i = threadIdx.x + (blockDim.x*4) * blockIdx.x;
	int i2 = i + blockDim.x;
	int i3 = i2 + blockDim.x;
	int i4 = i3 + blockDim.x;
	if (i<size){
		res[i] = vec1[i] + vec2[i];
	}
	if(i2<size){
		res[i2] = vec1[i2] + vec2[i2];
	}
	if(i3<size){
		res[i3] = vec1[i3] + vec2[i3];
	}
	if(i4<size){
		res[i4] = vec1[i4] + vec2[i4];
	}
}

// scale kernel:
__global__ void J_vec_scale(float* vec1, float alfa, unsigned int size){

	int i = threadIdx.x + (blockDim.x*4) * blockIdx.x;
	int i2 = i + blockDim.x;
	int i3 = i2 + blockDim.x;
	int i4 = i3 + blockDim.x;
	if (i<size){
		vec1[i] = vec1[i] * alfa;
	}
	if(i2<size){
		vec1[i2] = vec1[i2] * alfa;
	}
	if(i3<size){
		vec1[i3] = vec1[i3] * alfa;
	}
	if(i4<size){
		vec1[i4] = vec1[i4] * alfa;
	}
}

// sub kernel:
__global__ void J_vec_sub(float* vec1, float* vec2, float* res, unsigned int size){

	int i = threadIdx.x + (blockDim.x*4) * blockIdx.x;
	int i2 = i + blockDim.x;
	int i3 = i2 + blockDim.x;
	int i4 = i3 + blockDim.x;
	if (i<size){
		res[i] = vec1[i] - vec2[i];
	}
	if(i2<size){
		res[i2] = vec1[i2] - vec2[i2];
	}
	if(i3<size){
		res[i3] = vec1[i3] - vec2[i3];
	}
	if(i4<size){
		res[i4] = vec1[i4] - vec2[i4];
	}
}
```

```
// dot kernel
__global__ void j_vec_dot(float* a_in, float* b_in, float* c_out, int size){
    unsigned int tid = threadIdx.x;
    unsigned int bid = blockIdx.x;
    unsigned int blockSize = blockDim.x;
    SharedMemory<float> smem;
    float* sdata = smem.getPointer();

    sdata[tid] = 0.0f;
    __syncthreads();

    unsigned int i = tid + bid*blockSize*4;
    if(i+3*blockSize < size){
        sdata[tid] = a_in[i] * b_in[i] + a_in[i+blockSize] * b_in[i+blockSize] +
a_in[i+2*blockSize] * b_in[i+2*blockSize] + a_in[i+3*blockSize] * b_in[i+3*blockSize];
    } else if(i+2*blockSize < size){
        sdata[tid] = a_in[i] * b_in[i] + a_in[i+blockSize] * b_in[i+blockSize] +
a_in[i+2*blockSize] * b_in[i+2*blockSize];
    }else if(i+blockSize < size){
        sdata[tid] = a_in[i] * b_in[i] + a_in[i+blockSize] * b_in[i+blockSize];
    }else if(i< size){
        sdata[tid] = a_in[i] * b_in[i];
    }
    __syncthreads();
    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

        //unroll do ultimo warp
    if (tid < 32){
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
       sdata[tid] += sdata[tid +  1];
    }

    // escrever o resultado do bloco para memória partilhada
    if (tid == 0) c_out[blockIdx.x] = sdata[0];
}
```

Anexo B – Código estendido no CNC

```
// BCRS 3x3 multiplication kernel

__global__ void CNCMat3x3VecMult3Kernel ( float3 * mat0, float3 * mat1,
                              float3 * mat2, unsigned int size_matrix,
                              uint2 * rowptr, unsigned int size_rowptr,
                              unsigned int * colind, unsigned int size_colind,
                              float3 * x, float3 * b, unsigned int size_vec ) {

        const unsigned int index = compute_thread_index () ;

        if ( index*3 < size_vec ) {

                uint2 rowptr_bounds = rowptr[index] ;
                float3 res ;
                res.x=res.y=res.z= 0.0f ;

                unsigned int ci = 0 ;
                float3 x_vec ;
                float3 row0  ;
                float3 row1  ;
                float3 row2  ;

                for ( int i=rowptr_bounds.x; i<rowptr_bounds.y; i++ ) {
                        row0 = mat0[i] ;
                        row1 = mat1[i] ;
                        row2 = mat2[i] ;
                        ci = colind[i] ;
                        x_vec = x[ci]  ;
                        res.x += row0.x*x_vec.x+row0.y*x_vec.y+row0.z*x_vec.z ;
                        res.y += row1.x*x_vec.x+row1.y*x_vec.y+row1.z*x_vec.z ;
                        res.z += row2.x*x_vec.x+row2.y*x_vec.y+row2.z*x_vec.z ;

                }
                b[index] = res ;
        }
}


template<> inline void CNCSparseMatrixBCRS<float,3,3>::gpu_allocate_and_upload () {

        cublasStatus st;
        unsigned int size_matrix = a.size () ;
        st = cublasAlloc ( size_matrix/3+16, sizeof(float), &(gpu_mat0_) ) ;
        CNCgetError ( st ) ;
        st = cublasAlloc ( size_matrix/3+16, sizeof(float), &(gpu_mat1_) ) ;
        CNCgetError ( st ) ;
        st = cublasAlloc ( size_matrix/3+16, sizeof(float), &(gpu_mat2_) ) ;
        CNCgetError ( st ) ;
        float * m0 = (float*)malloc(sizeof(float)*(size_matrix/3)) ;
        float * m1 = (float*)malloc(sizeof(float)*(size_matrix/3)) ;
        float * m2 = (float*)malloc(sizeof(float)*(size_matrix/3)) ;
        {for(long i=0; i<(long)(size_matrix/9); i++) {
                m0[i*3  ] = a.data()[i*9   ] ;
                m0[i*3+1] = a.data()[i*9+ 1] ;
                m0[i*3+2] = a.data()[i*9+ 2] ;

                m1[i*3  ] = a.data()[i*9+ 3] ;
                m1[i*3+1] = a.data()[i*9+ 4] ;
                m1[i*3+2] = a.data()[i*9+ 5] ;

                m2[i*3  ] = a.data()[i*9+ 6] ;
                m2[i*3+1] = a.data()[i*9+ 7] ;
                m2[i*3+2] = a.data()[i*9+8] ;

        }}
```

```
        st = cublasSetVector ( size_matrix/3, sizeof(float), m0, 1, gpu_mat0_, 1 ) ;
        CNCgetError ( st ) ;
        st = cublasSetVector ( size_matrix/3, sizeof(float), m1, 1, gpu_mat1_, 1 ) ;
        CNCgetError ( st ) ;
        st = cublasSetVector ( size_matrix/3, sizeof(float), m2, 1, gpu_mat2_, 1 ) ;
        CNCgetError ( st ) ;

        free (m0) ;
        m0 = NULL;
        free (m1) ;
        m1 = NULL;
        free (m2) ;
        m2 = NULL;
        st = cublasAlloc ( colind.size()+16, 4, &(gpu_ci_) ) ;
        CNCgetError ( st ) ;
        st = cublasAlloc ( 2*(rowptr.size()-1)+16, sizeof(unsigned int), &(gpu_redundant_rp_) ) ;
        CNCgetError ( st ) ;

        if (DEBUG_) std::cout << "CNCallocate_and_copy" << std::endl;
        unsigned int * ci = CNCallocate_and_copy<long, unsigned int> (
colind.data(),colind.size() ) ;
        if (DEBUG_) std::cout << "CNC_allocate" << std::endl;
        uint2 * cpu_redundant_rp =  CNCallocate<uint2>(rowptr.size()-1) ;
        {for (unsigned int i=0; i<rowptr.size()-1; i++) {
                cpu_redundant_rp[i].x = rowptr.data()[i  ] ;
                cpu_redundant_rp[i].y = rowptr.data()[i+1] ;
        }}

        if (DEBUG_) std::cout << "Other copys to gpuMem" << std::endl;
        st = cublasSetVector ( rowptr.size()-1, sizeof(uint2), cpu_redundant_rp, 1,
gpu_redundant_rp_, 1 ) ;
        CNCgetError ( st ) ;
        st = cublasSetVector ( colind.size(), sizeof(unsigned int), ci, 1, gpu_ci_, 1 ) ;
        CNCgetError ( st ) ;

        if (DEBUG_) std::cout << "CNC_deallocates" << std::endl;
        CNCdeallocate<uint2> ( cpu_redundant_rp ) ;
        CNCdeallocate<unsigned int> ( ci ) ;
}

template<> inline void CNCSparseMatrixBCRS<float,3,3>::gpu_deallocate () {
        cublasStatus st;
        st = cublasFree ( (void*)gpu_ci_ ) ;
        CNCgetError ( st ) ;
        gpu_ci_ = NULL;
        st = cublasFree ( (void*)gpu_redundant_rp_ ) ;
        CNCgetError ( st ) ;
        gpu_redundant_rp_ = NULL;
        st = cublasFree ( (void*)gpu_mat0_ ) ;
        CNCgetError ( st ) ;
        gpu_mat0_ = NULL;
        st = cublasFree ( (void*)gpu_mat1_ ) ;
        CNCgetError ( st ) ;
        gpu_mat1_ = NULL;
        st = cublasFree ( (void*)gpu_mat2_ ) ;
        CNCgetError ( st ) ;
        gpu_mat2_ = NULL;
}

template<> inline void CNCSparseMatrixBCRS<float,3,3>::gpu_mult ( const void * x, void * y,
                                        const unsigned int vec_size ) {
        cnc_cuda_mat3x3vecmult3( (float3*)gpu_mat0_, (float3*)gpu_mat1_,
                                                (float3*)gpu_mat2_, a.size(),
                                                    (uint2*)gpu_redundant_rp_, rowptr.size(),
                                                    (unsigned int*)gpu_ci_, colind.size(),
                                                    (float3*)x, (float3*)y, vec_size ) ;

}
```

Anexo C – Código do MPCG em GPU

```
template<class T, class T1> inline int MPCGGPUSolver::solve_mpcg_internal (
                T &A,
                T1 &P,
                T1 &P_1,
                floatArray &b,
                floatArray &deltav,
                floatArray &dv0,
                floatArray &cf,
                T1 &SM,
                T1 &CSM,
                const unsigned int nb_iter_max,
                const double epsilon,
                const unsigned int block_size) {


        long N_Particles = deltav.size() ;
        cublasStatus st;

        //setting CNC grid configs:

        cnc_cuda_set_dim_vec ( (unsigned int)(sqrt((float)N_Particles)/THREAD_BLOCK_SIZE+1),
                                        (unsigned
int)(sqrt((float)N_Particles)/THREAD_BLOCK_SIZE+1),
                                        THREAD_BLOCK_SIZE, THREAD_BLOCK_SIZE ) ;

        cnc_cuda_set_dim_vec2( (unsigned int)(sqrt((float)N_Particles/2.0f)/THREAD_BLOCK_SIZE+1),
                                        (unsigned
int)(sqrt((float)N_Particles/2.0f)/THREAD_BLOCK_SIZE+1),
                                        THREAD_BLOCK_SIZE, THREAD_BLOCK_SIZE ) ;

        cnc_j_cuda_set_dim_vec3( (unsigned
int)(sqrt((float)N_Particles/3.0f)/THREAD_BLOCK_SIZE+1),
                                        (unsigned
int)(sqrt((float)N_Particles/3.0f)/THREAD_BLOCK_SIZE+1),
                                        THREAD_BLOCK_SIZE, THREAD_BLOCK_SIZE ) ;

        cnc_cuda_set_dim_vec4( (unsigned int)(sqrt((float)N_Particles/4.0f)/THREAD_BLOCK_SIZE+1),
                                        (unsigned
int)(sqrt((float)N_Particles/4.0f)/THREAD_BLOCK_SIZE+1),
                                        THREAD_BLOCK_SIZE, THREAD_BLOCK_SIZE ) ;

        // setting temporary device vector, N_Particles size, filled with zeros

        float* gpu_zeros = NULL;

        float* zeros = (float*)malloc(N_Particles*sizeof(float));
        for(int x=0;x<N_Particles;x++){
                zeros[x]=0.0f;
        }
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_zeros  );
        CNCgetError ( st ) ;
        cudaMemcpy((void**)gpu_zeros,(void**)zeros, N_Particles*sizeof(float),
cudaMemcpyHostToDevice);
        free(zeros);

        // mpcg internal vars:
        float* gpu_Pbf = NULL;
        float* gpu_r = NULL;
        float* gpu_Adv = NULL;
        float* gpu_q = NULL;
        float* gpu_alfac = NULL;
        float* gpu_c = NULL;
        float* gpu_s = NULL;
        float* gpu_deltav = NULL;
        float* gpu_b = NULL;
```

```
        float* gpu_dv0 = NULL;
        float* gpu_cf = NULL;
        float* gpu_tmp = NULL;

        //memory allocation
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_Pbf  );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_r  );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_Adv  );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_q );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_alfac );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_c );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_s );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_deltav );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_b );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_dv0 );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_cf );
        CNCgetError ( st ) ;
        st = cublasAlloc ( N_Particles, sizeof(float), (void**)&gpu_tmp );
        CNCgetError ( st ) ;
        st = cublasSetVector ( N_Particles, sizeof(float), dv0.data(), 1, gpu_dv0, 1 );
        CNCgetError(st);
        st = cublasSetVector ( N_Particles, sizeof(float), deltav.data(), 1, gpu_deltav, 1 );
        CNCgetError(st);
        st = cublasSetVector ( N_Particles, sizeof(float), b.data(), 1, gpu_b, 1 );
        CNCgetError(st);
        st = cublasSetVector ( N_Particles, sizeof(float), cf.data(), 1, gpu_cf, 1 );
        CNCgetError(st);

        // matrix allocation and upload
        A.gpu_allocate_and_upload() ;
        P.gpu_allocate_and_upload() ;
        P_1.gpu_allocate_and_upload() ;
        SM.gpu_allocate_and_upload() ;
        CSM.gpu_allocate_and_upload() ;

        float alfa, beta, deltaNew, deltaOld, delta0;

        //setting up timer to "time" the MPCG GPU METHOD:
        unsigned int timer = 0;
        float currTime = 0.0f;
        cutCreateTimer( &timer);
    cutStartTimer( timer);

////////////////////////////////////////////////////////////////////////////////////////
////                            START OF GPU MPCG METHOD:                           //////////
////////////////////////////////////////////////////////////////////////////////////////
        //Boxerman Code:
        //Filter(gpu_dv0);
        if(USE_CUBLAS){
                std::cout << "\tUsing nvidia CUBLAS vector operations" << std::endl;
        } else {
                if(USE_CUBLAS_DOT) std::cout<<"\tUsing Nvidia cublas dot\n";
                std::cout << "\tUsing J vector operations" << std::endl;
        }
        SM.gpu_mult( gpu_dv0, gpu_tmp, N_Particles );
        cudaMemcpy((void**)gpu_dv0, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
```

```
        //Filter(gpu_deltav);
        CSM.gpu_mult( gpu_deltav, gpu_tmp, N_Particles );
        cudaMemcpy((void**)gpu_deltav, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);

        CSM.gpu_deallocate();

        // AFTER BOXERMAN CODE
        //dv = dv+dv0
        if(USE_CUBLAS){
                cublasSaxpy(N_Particles, 1, gpu_dv0, 1, gpu_deltav, 1);
        }else{
                j_cuda_vec_add(gpu_deltav, gpu_dv0, gpu_tmp, N_Particles);
                cudaMemcpy((void**)gpu_deltav, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
        }
        // Adv = A*deltav
        A.gpu_mult ( gpu_deltav, gpu_Adv, N_Particles ) ;

        // r = b - Adv
        if(USE_CUBLAS){
                cudaMemcpy(gpu_r, gpu_Adv, (N_Particles)*sizeof(float), cudaMemcpyDeviceToDevice);
                cublasSscal ( N_Particles, -1, gpu_r, 1 ) ;
                cublasSaxpy ( N_Particles, 1.0f, gpu_b, 1, gpu_r, 1 ) ;
        }else{
                j_cuda_vec_sub(gpu_b, gpu_Adv, gpu_r, N_Particles);
        }

        //Filter(r);
        SM.gpu_mult(gpu_r, gpu_tmp, N_Particles);
        cudaMemcpy((void**)gpu_r, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);

        //Filter(b);
        SM.gpu_mult(gpu_b, gpu_tmp, N_Particles);
        cudaMemcpy((void**)gpu_b, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);

        //Pbf = P * b
        P.gpu_mult(gpu_b, gpu_Pbf, N_Particles);

        //dot(Pbf, b) = delta0
        if(USE_CUBLAS||USE_CUBLAS_DOT){
                delta0 = cublasSdot(N_Particles, gpu_Pbf, 1, gpu_b, 1);
        }else{
                cudaMemcpy((void**)gpu_tmp, (void**)gpu_zeros, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
                delta0 = j_cuda_dot(gpu_b, gpu_Pbf, gpu_tmp, N_Particles);
        }

        // c = P_1 * r
        P_1.gpu_mult(gpu_r, gpu_c, N_Particles);

        //Filter(c);
        SM.gpu_mult(gpu_c, gpu_tmp, N_Particles);
        cudaMemcpy((void**)gpu_c, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);

        //delta_new = r . c
        if(USE_CUBLAS||USE_CUBLAS_DOT){
                deltaNew = cublasSdot(N_Particles, gpu_r, 1, gpu_c, 1);
        }else{
                cudaMemcpy((void**)gpu_tmp, (void**)gpu_zeros, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
                deltaNew = j_cuda_dot(gpu_r, gpu_c, gpu_tmp, N_Particles);
        }

        unsigned int i = 0;
        //float gpu_w_time = 0.0f;
        while((i < nb_iter_max) && (deltaNew > epsilon*epsilon*delta0)){
```

```
		// q = A * c
		A.gpu_mult(gpu_c, gpu_q, N_Particles);

		//Filter(q);
		SM.gpu_mult(gpu_q, gpu_tmp, N_Particles);
		cudaMemcpy((void**)gpu_q, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);

		//alfa = deltaNew/dot(c, q)
		if(USE_CUBLAS||USE_CUBLAS_DOT){
			alfa = cublasSdot(N_Particles, gpu_c, 1, gpu_q, 1);
		}else{
			cudaMemcpy((void**)gpu_tmp, (void**)gpu_zeros,
(N_Particles)*sizeof(float), cudaMemcpyDeviceToDevice);
			alfa = j_cuda_dot(gpu_c, gpu_q, gpu_tmp, N_Particles);
		}

		alfa = deltaNew/alfa;

		//alfac = c
		cudaMemcpy((void**)gpu_alfac, (void**)gpu_c, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);

		//alfac = alfac * alfa
		if(USE_CUBLAS){
			cublasSscal(N_Particles, alfa, gpu_alfac, 1);
		}else{
			j_cuda_vec_scale(gpu_alfac, alfa,N_Particles);
		}

		//deltav = deltav + alfac
		if(USE_CUBLAS){
			cublasSaxpy( N_Particles, 1.0f, gpu_alfac, 1, gpu_deltav, 1 );
		}else{
			j_cuda_vec_add(gpu_deltav, gpu_alfac, gpu_tmp, N_Particles);
			cudaMemcpy((void**)gpu_deltav, (void**)gpu_tmp,
(N_Particles)*sizeof(float), cudaMemcpyDeviceToDevice);
		}

		//q = q*alfa
		if(USE_CUBLAS){
			cublasSscal(N_Particles, alfa, gpu_q, 1);
		}else{
			j_cuda_vec_scale(gpu_q,alfa,N_Particles);
		}

		if (DEBUG_) std::cout << "r = r - q" << std::endl;
		// r = r - q
		if(USE_CUBLAS){
			cublasSaxpy ( N_Particles, -1.0f, gpu_q, 1, gpu_r, 1 ) ;
		}else{
			j_cuda_vec_sub(gpu_r, gpu_q, gpu_tmp, N_Particles);
		cudaMemcpy((void**)gpu_r, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
		}

		// P_1 * r = s
		P_1.gpu_mult(gpu_r, gpu_s, N_Particles);

		// deltaOld = deltaNew
		deltaOld = deltaNew;

		//deltaNew = dot(r, s)
		if(USE_CUBLAS||USE_CUBLAS_DOT){
			deltaNew = cublasSdot(N_Particles, gpu_r, 1, gpu_s, 1);
		}else{
			cudaMemcpy((void**)gpu_tmp, (void**)gpu_zeros,
(N_Particles)*sizeof(float), cudaMemcpyDeviceToDevice);
			deltaNew = j_cuda_dot(gpu_r, gpu_s, gpu_tmp, N_Particles);
```

```
                }

                //beta=deltaNew/deltaOld
                beta = deltaNew/deltaOld;

                //c = c*beta
                if(USE_CUBLAS){
                        cublasSscal(N_Particles, beta, gpu_c, 1);
                }else{
                        j_cuda_vec_scale(gpu_c, beta, N_Particles);
                }

                //c = c + s
                if(USE_CUBLAS){
                        cublasSaxpy(N_Particles, 1.0f, gpu_s, 1, gpu_c, 1);
                }else{
                        j_cuda_vec_add(gpu_c, gpu_s, gpu_tmp, N_Particles);
                        cudaMemcpy((void**)gpu_c, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
                }

                if (DEBUG_) std::cout << "Filter(c)" << std::endl;
                //Filter(c)
                SM.gpu_mult(gpu_c, gpu_tmp, N_Particles);
                cudaMemcpy((void**)gpu_c, (void**)gpu_tmp, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
                i++;

        }
        if (DEBUG_) std::cout << "deltav = A * deltav;" << std::endl;
        //Adv = A * deltav;
        A.gpu_mult(gpu_deltav, gpu_Adv, N_Particles);

        //cf = b - Adv
        if(USE_CUBLAS){
                cudaMemcpy((void**)gpu_cf, (void**)gpu_Adv, (N_Particles)*sizeof(float),
cudaMemcpyDeviceToDevice);
                cublasSscal(N_Particles, -1, gpu_cf, 1);
                cublasSaxpy(N_Particles, 1, gpu_b, 1, gpu_cf, 1);
        }else{
                j_cuda_vec_sub(gpu_b, gpu_Adv, gpu_cf, N_Particles);
        }
        cudaThreadSynchronize();
        cutStopTimer(timer);
    float gpu_mpcg_solve_time = cutGetTimerValue(timer);
    cutDeleteTimer(timer);
    std::cout << "\tGPU mpcg solver execution time = " <<gpu_mpcg_solve_time<<" (ms)\n\n";

        //copy deltav & cf back to CPU:
        st = cublasGetVector ( N_Particles, sizeof(float), gpu_deltav, 1, deltav.data(), 1 ) ;
        CNCgetError ( st ) ;
        st = cublasGetVector ( N_Particles, sizeof(float), gpu_cf, 1, cf.data(), 1 ) ;
        CNCgetError ( st ) ;
        st = cublasGetVector ( N_Particles, sizeof(float), gpu_b, 1, b.data(), 1 ) ;
        CNCgetError ( st ) ;
        st = cublasGetVector ( N_Particles, sizeof(float), gpu_dv0, 1, dv0.data(), 1 ) ;
        CNCgetError ( st ) ;

        //free gpu memory;
        st = cublasFree (gpu_zeros ) ;
        CNCgetError ( st ) ;
        gpu_zeros = NULL;
        st = cublasFree(gpu_Pbf) ;
        CNCgetError ( st ) ;
        gpu_Pbf = NULL;
        st = cublasFree(gpu_r) ;
        CNCgetError ( st ) ;
        gpu_r = NULL;
        st = cublasFree (gpu_Adv  ) ;
        CNCgetError ( st ) ;
```

```
        gpu_Adv = NULL;
        st = cublasFree (gpu_q ) ;
        CNCgetError ( st ) ;
        gpu_q = NULL;
        st = cublasFree (gpu_alfac ) ;
        CNCgetError ( st ) ;
        gpu_alfac = NULL;
        st = cublasFree (gpu_c ) ;
        CNCgetError ( st ) ;
        gpu_c = NULL;
        st = cublasFree (gpu_s ) ;
        CNCgetError ( st ) ;
        gpu_s = NULL;
        st = cublasFree (gpu_deltav ) ;
        CNCgetError ( st ) ;
        gpu_deltav = NULL;
        st = cublasFree (gpu_b ) ;
        CNCgetError ( st ) ;
        gpu_b = NULL;
        st = cublasFree (gpu_dv0 ) ;
        CNCgetError ( st ) ;
        gpu_dv0 = NULL;
        st = cublasFree (gpu_cf ) ;
        CNCgetError ( st ) ;
        gpu_cf = NULL;
        st = cublasFree (gpu_tmp ) ;
        CNCgetError ( st ) ;
        gpu_tmp = NULL;
        if (DEBUG_) std::cout << "FREE GPU MATRIXES" << std::endl;

        A.gpu_deallocate () ;
        P.gpu_deallocate();
        P_1.gpu_deallocate();
        SM.gpu_deallocate();

        if (DEBUG_) std::cout << "END OF MPCG GPU INTERNAL SOLVE" << std::endl;

        return i;
}
```